# De la programmation parallèle structurée à la programmation pour la grille

Françoise Baude

U N I V E R S I T É  D E  N I C E  S O P H I A - A N T I P O L I S

# De la programmation parallèle structurée à la programmation pour la grille

Mémoire de Synthèse présenté à l'Université de Nice Sophia-Antipolis
pour l'obtention d'une

HABILITATION À DIRIGER LES RECHERCHES

Spécialité Informatique

par

**Françoise Baude-Dreysse**

Equipe OASIS

Equipe commune CNRS I3S - Univ. Nice Sophia Antipolis - INRIA Sophia Antipolis

Soutenue le 15 Septembre 2006
devant la commission d'examen composée de MM. :

| | | |
|---|---|---|
| Président : | M. Riveill | I3S-CNRS, UNSA |
| Rapporteurs : | M. Danelutto | Univ. of Pisa |
| | P. Kuonen | Ecole d'ingénieurs de Fribourg |
| | P. Lalanda | LSR IMAG, UJF Grenoble |
| Examinateur : | D. Caromel | INRIA, I3S-CNRS, UNSA, IUF |

# Avant-propos

Ce mémoire présente quelques quinze années de recherches, depuis ma thèse de doctorat en Décembre 1991 et mon recrutement en tant que maître de conférences en Octobre 1993 à l'Université de Nice Sophia-Antipolis. Ces recherches se sont déroulées au sein du Laboratoire I3S (Université de Nice Sophia Antipolis-C.N.R.S. UMR 6070), successivement dans l'équipe PACOM, puis de 1995 à 1998 dans le cadre du projet SLOOP, commun avec l'INRIA Sophia Antipolis, et enfin depuis 1999 dans le cadre du projet OASIS, également commun avec l'INRIA Sophia-Antipolis.

A l'heure des remerciements, mes premières pensées se dirigent sans hésitation vers les jeunes chercheurs (doctorants, ingénieurs, stagiaires de master) que j'ai été amenée à encadrer ou co-encadrer, certains d'entre eux étant depuis devenus des collègues. Leur montrer la voie que nous voulions explorer, leur permettre d'acquérir la compréhension nécessaire pour qu'ensuite nous puissions l'exploiter *ensemble*, ne serait-ce que pour un temps, fut pour moi un plaisir sans cesse renouvelé. La collaboration fructueuse qui se construit petit à petit avec Ludovic Henrio chercheur CNRS dans l'équipe depuis octobre 2005, est également précieuse. Mes remerciements vont aussi aux personnes dont quelques minutes d'attention, un ou deux conseils éclairés en matière d'orientation ou d'organisation m'ont permis d'arriver à mener ce travail à terme, sachant que le plus difficile fut de le mener en grapillant ça ou là quelques heures entre les tâches plus urgentes : Laurence Rideau, Jean-Marc Fédou, Serge Chaumette, Laure Blanc-Féraud, Didier Parigot, Jean-Claude Bermond.

C'est vers mon plus proche collaborateur depuis mon arrivée à l'Université de Nice Sophia-Antipolis en 1993, Denis Caromel, que vont toute ma gratitude et mon amitié. A lui revient le mérite de nous avoir engagés sur des pistes de recherche pertinentes dans cette thématique de la programmation par objets répartie dans le cadre du *grid computing*. Qu'il soit assuré de mon infaillible engagement et soutien dans l'accomplissement de notre métier d'enseignant-chercheur, ainsi que dans la plus difficile des épreuves qu'il soit donné de surmonter, celle de la disparition de sa femme et ses enfants.

Mes pensées et remerciements vont également à ma tant regrettée directrice de recherche, Isabelle Attali, emportée par le tsunami de décembre 2004 ainsi que ses deux petits garçons, Ugo et Tom, du même âge environ que les

deux miens. Ses encouragements à tous les membres de l'équipe, et nos amicales relations avaient crées depuis l'avènement du projet OASIS, un terrain idéal à l'accomplissement des travaux de recherche autour de la programmation répartie et de ProActive, qui constituent le sujet principal de cette HDR. Je suis sûre que sa confiance à mon égard, la joie de vivre qui était la sienne, et son dynamisme n'ont cessé de m'accompagner dans l'aboutissement de cette HDR, même depuis l'au delà. Je me sens en partie héritière de son enthousiasme à monter des collaborations qu'elles soient nationales, ou internationales, les faire aboutir, puis les faire vivre et qu'elles réussissent. L'ambition est de promouvoir les résultats de recherche de l'équipe, les faire évoluer, les consolider, et ce par le biais de leur partage au sein de tels partenariats, et de leur confrontation avec les besoins du monde industriel.

Enfin, c'est avec grand plaisir que je remercie très sincèrement mes rapporteurs et tous les membres de mon jury d'Habilitation, pour ce temps précieux qu'ils ont bien voulu consacrer à l'évaluation de mes travaux.

*A mon époux, Jean-Christophe, et nos deux garçons, Guillaume et Gabriel, qui tout en me soutenant dans mon activité professionnelle, réussissent dans le même temps à me prévenir d'un engloutissement complet dans cette même activité ...*

# Table des matières

# Table des figures

# Première partie

# Synthèse des Travaux

# Chapitre 1

# Introduction

## 1.1 Domaine de recherche

Notre activité de recherche se situe dans les domaines suivants :
- programmation parallèle *haut-niveau*, dans un cadre réparti, asynchrone, hétérogène (typiquement, grilles de calcul)
- intergiciels supports dans un tel cadre (utilisation, conception, développement)
- administration de plateformes réparties (application de ces concepts).

Plus globalement, nous nous intéressons aux modèles, langages, outils de développement et d'exécution (programmation, intégration, déploiement, supervision), pour applications réparties requérant un certain niveau de performance.

## 1.2 Objectifs

La problématique récurrente qui constitue notre objectif est *comment concilier la facilité d'utilisation des langages, des outils, tout en masquant l'hétérogénéité des supports d'exécution répartis, sans pour autant sacrifier ni le degré d'expressivité, ni les performances.*

Les infrastructures réparties qui constituent les supports d'exécution des applications sont en constante évolution (pour une vision synthétique de l'évolution de ces infrastructures et des problématiques de recherche associées à leur exploitation, on peut se référer à l'éditorial de l'ouvrage collectif que j'ai coordonné [E2][1]) :

[1] les références du type [Ex], [Jx], [Bx], etc., correspondent à ma liste de publications

- depuis les machines parallèles dédiées puis généralistes,
- en passant par les réseaux locaux de stations de travail et de PCs,
- les grappes de machines bâties autour d'un réseau de communcication si possible haute performance et propre à la grappe,
- jusqu'aux grilles de calcul, ce qui rajoute au moins deux caractéristiques importantes : l'interconnexion via un réseau longue distance et plusieurs domaines d'administration, d'où des performances de calcul et de communication hétérogènes, et des besoins de protection accrus.

Comparativement au cas séquentiel, il est évident qu'il est plus complexe de concevoir, programmer, déployer, surveiller une application qui met en jeu plusieurs activités concurrentes, éventuellement avec l'objectif qu'elles s'exécutent en parallèle pour plus de performance.

Il est également admis que, plus le niveau d'abstraction des outils proposés à l'utilisateur final (programmeur, ou administrateur) est élevé, plus le nombre de détails techniques logiciels ou matériels qui peuvent lui être masqués devient grand, et plus le potentiel de portabilité (et de pérennité) de ses réalisations est élevé. Cependant, toute la complexité inhérente au cas réparti se reporte sur les couches intermédiaires, situées entre les outils proposés à l'utilisateur et l'infrastructure cible. Maitriser la complexité inhérente au sein de ces couches devient donc un réel défi, qui passe en général par des compromis. Pour l'utilisateur final ce compromis se traduit en général par l'obligation de se confiner dans un cadre d'utilisation prédéfini et plus ou moins contraignant. Le système intermédiaire devant prendre en charge les applications ainsi produites, c'est-à-dire le système opératoire (runtime) aidé ou non par des transformateurs tels des compilateurs, doit être conçu afin de concilier performances et généricité pour plus de portabilité. Nous pensons que tout l'enjeu est de parvenir à rendre le cadre d'utilisation le moins strict possible et le plus aisé possible, tout en répondant aux soucis de portabilité, réutilisabilité, efficacité.

En résumé, le tryptique suivant délimite bien où notre recherche se situe et quel est le compromis que l'on explore : Puissance d'expression, Portabilité, Efficacité. C'est pour avoir une chance d'apporter des réponses satisfaisantes dans la recherche de ce type de compromis, que nous privilégions le paradigme de programmation à objets et à composants logiciels répartis,

---

classée chronologiquement, donnée dans le chapitre 2 de l'annexe, et reprise en partie sur ma page web `www.inria.fr/oasis/Francoise.Baude`. Avec la signification : J : journal, B : chapitre de livre, C : conférence avec sélection et actes, W : conférence ou atelier, sans actes ou à diffusion plus restreinte, avec sélection ou invitation R : rapport ou livrable.

ces objets ou composants étant déployées et interagissant grâce à des intergiciels appropriés. La suite du document argumentera bien évidemment un tel choix.

## 1.3 Démarche

L'arrêté et la circulaire relatifs à l'"Habilitation à Diriger des Recherches" (23 Novembre 1988, et 25 Avril 2002), définissent un certain nombre de critères d'évaluation du candidat quant à son parcours de chercheur, dont "*son aptitude à maîtriser une stratégie de recherche dans un domaine scientifique ou technologique suffisamment large et de sa capacité à encadrer de jeunes chercheurs*" [2].

Nous travaillons tout d'abord au niveau des langages et bibliothèques de programmation, qui permettent d'exprimer du parallélisme et dont la mise en œuvre se prête naturellement à des plateformes réparties, le cas échéant, hétérogènes ou à grande échelle. C'est pour cette raison que nous ne nous intéressons pas a priori à l'étude des langages qui requièrent pour leur mise en œuvre l'utilisation d'une mémoire partagée, physique ou virtuelle. Notre démarche consiste en quelque sorte à inciter le programmeur à *guider* la mise en œuvre ultérieure en environnement réparti, et ce en lui fournissant un cadre associé avec un certain nombre de mécanismes :

- dont l'utilisation soit éventuellement implicite, et ce sans aucune modification du langage de base lui-même
- dont l'exploitation efficace soit possible
- dont l'utilisation corresponde à un réel besoin applicatif ou amène un gain certain au moment de l'exécution (par exemple, en améliorant les performances, ou en permettant simplement d'offrir un service de type non-fonctionnel comme la sécurité, la tolérance aux pannes, l'équilibrage de charge, etc).

Dans le même temps, nous nous intéressons aux mécanismes et outils (typiquement, des intergiciels) permettant de déployer ces applications (si possible sur une large gamme de plateformes pour une bonne portabilité), les superviser, les adapter, les reconfigurer.

[2] Les autres critères étant la "*la reconnaissance du haut niveau scienfitique du candidat*", et "*le caractère original de sa démarche dans un domaine de la science.*

Finalement, l'objectif de ces recherches étant de faciliter la programmation efficace de plateformes réparties qui sont présentes dans le milieu de l'entreprise (au sens large), il nous semble important d'avoir un lien fort avec le monde réel. Ainsi, nous nous attachons à nous investir dans des collaborations contractuelles et/ou industrielles, qui nous permettent de mettre en évidence la pertinence de notre approche sur des problématiques ou applications réalistes, et comparer nos résultats avec d'autres utilisables dans ces mêmes contextes.

## 1.4 Résumé

L'axe de recherche suivi, qui a comme objectif de faciliter la programmation et la mise en œuvre d'applications parallèles et réparties, peut être présenté synthétiquement selon un espace défini selon les trois dimensions (voir figure 1.1) :

1. propriétés non-fonctionnelles que l'on se doit d'offrir
2. paradigmes de programmation adaptés pour la prise en compte de telles propriétés
3. degré d'automatisation ou de sophistication des outils requis pour pouvoir rendre possible la mise en œuvre des applications sur les plateformes réparties cibles.

Les contributions et travaux en cours s'orientent selon deux *angles d'attaque*, complémentaires et concourant à cet objectif : l'un consiste à proposer des extensions aux paradigmes de programmation parallèle et répartie. En particulier, on se donne comme objectif que les applications obtenues soient portables et entre autre, puissent s'exécuter sur grilles de calcul, et plus généralement sur tout support réparti constitué de la mise en réseau d'équipements hétérogènes. L'autre consiste à démontrer le potentiel et l'intérêt à utiliser ces paradigmes et leurs extensions apportées.

Selon le premier angle, nous nous basons principalement sur une méthodologie de programmation par objets actifs et composants logiciels répartis, et nous avons essentiellement contribué à leur extension pour du parallélisme : recouvrement calcul communication [J3,J5], gestion de la mobilité [J4] et du placement des activités [B1,B2,C20], groupes parallèles d'activités [B1,C17], composants répartis et parallèles [C12,B1]. Nous pensons que le paradigme par composants logiciels (qui de plus peut constituer le matériau de base

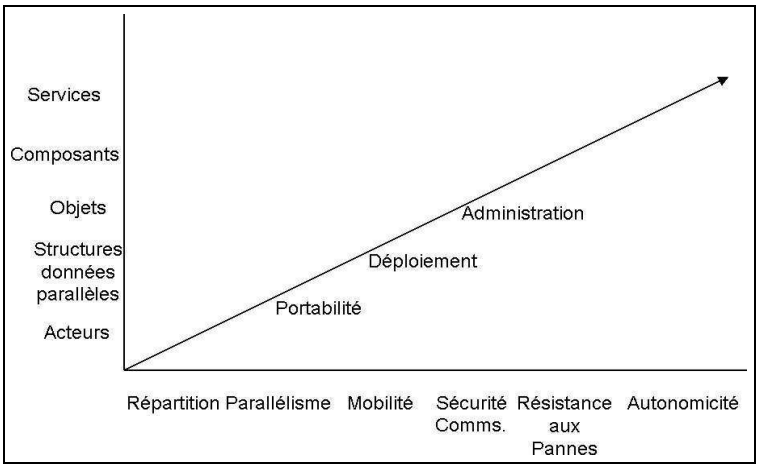



FIG. 1.1 – Structuration de l'espace des recherches

pour la programmation de services [B3]) est suffisamment riche pour pouvoir intégrer les qualités d'autonomicité devenues incontournables pour la maitrise des plateformes cibles d'aujourd'hui et de demain : plateforme globale complètement ouverte, bâtie sur Internet, qui offre à ses utilisateurs des services dont la nature relève de la combinaison de multiples fonctionnalités de type communication, calcul, stockage. Nous orientons donc notre travail vers l'introduction de qualités autonomes au sein des composants logiciels répartis [W22] (tel que nous l'explorons dans le projet européen de type IP, Bionets).

Le second angle d'attaque consiste à élargir les perspectives d'utilisation de ce type de programmation répartie par objet et composant. Un effort est fait quant aux types d'application. Au lieu de ne considérer que des champs d'application précis, par exemple, le calcul dans le domaine des sciences ([C13] revisité par une approche tout composant dans le projet ANR DiscoGrid [R18]), ou de la finance (projet ANR GCPMF), ...on s'intéresse également à des applications qui peuvent constituer des outils d'intêrét général ; par exemple, des plateformes d'administration système et réseau [C9,C11], des outils d'approvisionnement (déploiement) de services sur un parc d'équipements (projet RNRT PISE [W23]). Les outils ciblés par d'autres de nos travaux en démarrage, relèvent de : la mise en œuvre de services de communication répartis sur réseau mobile (projet Bionets), la *gridification* de serveurs et de bus d'entreprise (projets ITEA S4ALL, ARC

Automan et industriel AGOS).

Ainsi les perspectives d'utilisation de notre approche, et de l'environnement associé ObjectWeb ProActive, pour la programmation et la mise en œuvre parallèle et répartie concernent n'importe quelle application résultant de l'interaction d'entités logicielles réparties. De plus, nous pourrions même envisager que certaines de ces applications, qui sont plutôt considérées comme des outils, puissent enrichir l'environnement lui-même. Par exemple, un outil d'approvisionnement de composants logiciels, efficace et robuste car réalisé par le biais d'objets actifs, mobiles et parallèles, pourrait être utile afin de déployer sur des parcs de grande taille, des applications développées pour cet environnement. De même, un outil de supervision efficace et passant bien à l'échelle, réalisé selon ce paradigme à objet actif et composant, pourrait être intégré à l'environnement en vue de son administration lorsque il est utilisé à grande échelle.

## 1.5 Organisation du mémoire

Ce mémoire s'attache à présenter une synthèse de nos travaux depuis 1991 (année de mon doctorat). Nous allons essayer de mettre en avant la cohérence thématique de notre activité de recherche.

Nous éviterons, autant que possible, de paraphraser le contenu de nos publications ou des manuscrits de doctorats (co)-encadrés ...Nous serons plutôt à la recherche d'un éclairage nouveau et d'une articulation entre les différents thèmes.

Dans la première partie de cette synthèse (Chapitre 2) nous abordons les aspects liés aux *langages de programmation parallèle* exploitant essentiellement le parallélisme dit de données (ou de résolution). Cette synthèse repose sur des contributions personnelles qui ne sont pas récentes. Néanmoins, un effort est fait pour les situer au vu de travaux plus récents au sein de la communauté. Elle peut donc éventuellement être sautée ! Cependant, elle a l'avantage de mettre en lumière les problématiques et orientations choisies alors, et qui éclairent donc mieux nos thèmes de recherches actuels.

Le Chapitre 3 est dédié à nos travaux autour du concept à objet actif. Nous montrons dans quelles directions nous avons contribué à fournir des extensions à ce concept afin de structurer la gestion du parallélisme et de la répartition. Nous revenons sur différents mécanismes et concepts introduits

afin de : recouvrir calcul et communication, permettre aux objets d'être mobiles, organiser les objets en groupes et communiquer efficacement avec eux. Nous décrivons ensuite les efforts accomplis pour rendre portable le support d'exécution de ces objets. Nous terminons par la présentation de l'extension de cette approche objet pour programmer des applications parallèles et réparties à l'aide de composants logiciels.

Le chapitre 4 est dédié à l'illustration de l'utilisation des concepts introduits dans le chapitre 3, dans des domaines d'application autres que ceux où la programmation parallèle est habituellement utilisée. Nous pensons que la prolifération des équipements en réseau et donc celle des applications ou services qu'ils proposent, nécessite l'usage de concepts et de techniques issus du parallélisme. Nous étudions l'utilisation d'objets actifs mobiles pour la supervision système et réseau. Nous montrons ensuite comment les groupes d'objets actifs peuvent être utiles pour la gestion de parcs d'équipements de grande taille.

En guise de conclusion et perspectives, le chapitre 5 présente l'ensemble des axes de recherche en cours. Chacun de ces axes constitue un prolongement naturel des recherches présentées dans ce mémoire : programmation parallèle orientée objet, composants parallèles et répartis, interopérabilité des systèmes de grille, déploiement et supervision. Comme preuve de l'effort constant pour contribuer aux travaux de la communauté académique ou industrielle, nous insistons sur les partenariats et contrats qui soutiennent ces axes.

La Partie II regroupe les publications qui servent de support à ce mémoire. Suivent un curriculum vitae, une bibliographie personnelle, puis générale.

# Chapitre 2

# Programmation parallèle structurée

Nos premiers travaux de recherche débutés en 1988 étaient centrés autour de la comparaison des puissances d'expression et d'efficacité d'implantation du modèle acteur et du modèle PRAM [C2], dans l'optique d'exploiter le *parallélisme de données*. Une vision succinte de ces travaux, poursuivis durant notre post-doctorat est donnée dans l'article [Section 2.1.1, page 95][1] . Cet article motive l'intérêt d'une approche de programmation parallèle à base de squelettes, constituant un bon compromis pour une exploitation maitrisée des machines et supports de calcul à parallélisme massif, et donc centrée sur le parallélisme que l'on peut extraire en présence d'une quantité importante de données à traiter. Ce qui nous a motivé est donc une approche parallèle structurante, articulée autour des structures de données globales (liste, arbre, etc), offrant un fort potentiel de réutilisation (opérateurs génériques sur de telles structures de données), et dans le même temps, un fort potentiel de mise en œuvre efficace.

Ce chapitre essaye de résumer et positionner de manière aussi synthétique que possible ces recherches ; nous nous efforcerons surtout de donner tous les éléments permettant de comprendre la démarche, qui depuis notre arrivée au laboratoire I3S nous a amenée à recentrer/focaliser nos recherches actuelles autour d'une approche orientée objet pour la programmation parallèle et répartie.

[1] les références du type [Section 2.1.4, page 90] indiquent un article inclus dans la partie II du manuscrit, Chapitre 2. Une notation indexée de la forme [Section 2.1.4, page 90*+14*] permet une reférence à une page donnée de l'article.

## 2.1 Programmation parallèle en langage acteur

La question principale posée puis résolue par notre travail de doctorat était la suivante : *La programmation par acteurs (tâches asynchrones, concurrentes, échangeant uniquement des messages) peut-elle être parallèle ?*

### 2.1.1 Architectures de machines parallèles

Dans ces années fin 1980, débuts des années 1990, les machines parallèles à *mémoire physique répartie* se répandaient. La raison principale pour disposer d'une mémoire physique répartie et non centralisée, venait du souhait d'interconnecter un nombre de CPUs toujours plus important autour d'un réseau d'interconnexion rapide, à faible diamètre (distance maximum entre n'importe quelle paire de processeurs) et suffisamment riche en liens de communication pour permettre du parallélisme dans la gestion d'un nombre important de communications simultanées. Se posait la question du placement des bancs mémoire. Deux familles de solutions se présentaient, que l'on peut schématiquement exposer ainsi :

- les bancs mémoire et les CPUs sont placés de part et d'autre du réseau d'interconnexion. Toute requête d'accès à la mémoire a donc la même latence minimum, qui est le temps de traversée du réseau de part et d'autre, exprimé en général de manière proportionnelle au nombre d'étages intermédiaires constitués de mini-routeurs à traverser [2]
- chaque processeur dispose de sa propre mémoire locale, la paire constituant un nœud. C'est par le biais de ses capacités de communication avec des nœuds voisins qu'un processeur peut obtenir une donnée de n'importe quel banc mémoire situé sur un des nœuds du réseau. La notion de voisinage physique devient alors importante, puisque les temps d'accès à une donnée sur la machine sont proportionnels à la distance qui sépare deux nœuds, distance qui n'est donc pas toujours égale au diamètre.

[2] un exemple typique est le réseau *butterfly* constitué de N CPUs, N bancs de mémoire, log N étages de N routeurs 2x2 chacun, les sorties de chaque routeur d'un étage donné étant reliées respectivement au routeur de l'étage suivant ayant la même position et au routeur de l'étage suivant dont l'éloignement est deux fois plus grand à chaque fois.

En terme de conception d'architecture matérielle, la deuxième catégorie est apparue historiquement après la première, du fait de sa plus grande extensibilité : à condition que le nombre de liens d'interconnexion par nœud reste constant, il suffit de brancher de nouveaux nœuds à ceux qui n'ont pas encore tous leurs voisins, pour augmenter les capacités de traitement de la machine parallèle. Au contraire, dans le premier cas, rajouter de nouveaux CPUs et bancs de mémoire, requiert de repenser l'architecture complète du réseau de routeurs. L'archétype d'un nœud permettant de construire des machines parallèles rentrant dans la seconde catégorie est le Transputer [41] disposant d'un CPU et de mémoire, ainsi que de 4 liens de communications, servant de brique de base, type Lego, pour bâtir des architectures en forme de cubes (grilles de dimension 3) (par ex. la J-machine [61]).

### 2.1.2 Modèle PRAM et machines parallèles à mémoire globale

La première catégorie de machine parallèle, où le temps d'accès à la mémoire est *uniforme*, a naturellement conduit à privilégier l'utilisation d'un paradigme de programmation parallèle fondé sur une mémoire commune.

Le modèle théorique de calcul parallèle à mémoire commune, c'est-à-dire le modèle PRAM [38], extension naturelle du modèle RAM, s'avérait donc un candidat adapté pour l'expression et l'évaluation d'algorithmes parallèles. La classe de problèmes pour lesquels il existe un algorithme PRAM permettant de les résoudre pour une donnée d'entrée de taille N, en temps parallèle de l'ordre de O($log^k N$) en utilisant un nombre de processeurs de l'ordre de O(N) regroupe les problèmes qu'on sait résoudre rapidement en parallèle. Cette classe de complexité se nomme $NC$ (et plus précisément $NC_{PRAM}$ dans la suite). De plus, $NC_{PRAM}$ contient des problèmes dont la résolution parallèle est optimale, c.a.d. ayant une efficacité en O(1). L'efficacité se mesure par le rapport du nombre de processeurs nécessaires, multiplié par le temps de résolution en parallèle, sur le temps de résolution séquentiel.

La sémantique opérationnelle du modèle de calcul PRAM peut être qualifiée de SIMD (Single Instruction Multiple Data), selon la classification bien connue de Flynn. Cependant, dans le modèle PRAM, il est fait l'hypothèse qu'une instruction d'accès à n'importe quelle adresse en mémoire globale commune a un temps d'exécution similaire à n'importe quelle autre instruction (une instruction d'accès à la mémoire commune par n'importe quel pro-

cesseur, en écriture ou en lecture s'exécute en temps O(1)[3]).

Avec l'apparition de machines parallèles de la première catégorie, il s'est avéré utile d'inventer des protocoles d'accès simultanés à la mémoire globale commune qui, malgré la présence d'un réseau d'interconnexion à traverser, ne modifient pas la complexité des problèmes. L'objectif a été atteint : il existe de nombreux protocoles de communication de N messages représentant N requêtes d'accès en lecture ou en écriture à la mémoire commune, capables de router ces messages en un temps parallèle O(log N), avec une forte probabilité [68, 82]. Ainsi, si un problème est dans $NC_{PRAM}$ il y reste même lorsque il est résolu autrement que de manière théorique, c'est-à-dire sur une machine parallèle !

De plus ces protocoles permettent une émulation *efficace* du modèle PRAM sur une véritable machine disposant de P processeurs, P étant de l'ordre de N/log N. Dans ce cas, chaque processeur physique simule O(log N) processeurs PRAM. Il faut cependant que le réseau d'interconnexion soit assez riche en routeurs ou en liens de communication pour supporter l'acheminement de ces O(N) messages en temps maximum O(log N). C'est le cas des machines parallèles de la première catégorie : le nombre total de routeurs est d'un facteur logarithmique plus élevé que le nombre de processeurs.

Intuitivement, il y a suffisamment de place (moyens matériels) dans le réseau d'interconnexion pour que le protocole de communication arrive à répartir tous les messages à router sans créer de contention. Par contre, ce n'est pas le cas de la seconde catégorie de machines parallèles, notamment celles pour lesquelles le nombre de liens de communication par nœud est une constante, quel que soit le nombre total de nœuds.

### 2.1.3 Modèle à passage de message et machines parallèles à mémoire totalement répartie

La seconde catégorie de machine parallèle, où tous les bancs mémoire sont en quelque sorte privés à chaque nœud, a naturellement conduit à l'utilisation d'un paradigme de programmation parallèle privilégiant l'échange d'information avec les voisins directs de chaque nœud. Des caractéristiques d'une

[3]dans la variante CRCW (Concurrent Read, Concurrent Write) du modèle PRAM, et sous certaines politiques de gestion des conflits en écriture ; à l'opposé, la variante EREW (Exclusive Read, Exclusive Write) requiert en plus de prendre en compte dans l'algorithme le fait que plusieurs accès concurrents à la même adresse en mémoire globale sont impossibles.

telle architecture physique découle donc un paradigme de programmation parallèle fondé sur l'expression de tâches parallèles asynchrones, communicant par échange explicite de messages, tâches si possible placées sur des nœuds voisins. Les langages de programmation par acteur [5] sont justement fondés sur un tel paradigme. Dans sa version la plus épurée, un acteur est une tâche asynchrone, autonome, qui interagit avec les autres acteurs qui font partie de ses connaissances (pour lesquels il a une référence) par envoi ou réception asynchrone de messages.

> La problématique que nous nous sommes posés alors était la suivante : le parallélisme de tâches (exprimé par exemple via un langage acteur) semble naturel pour exploiter le parallélisme physique de telles architectures. Mais, qu'en est-t-il pour le parallélisme de données ?

Pour cela, on a commencé à définir un modèle de calcul parallèle, que l'on a nommé *modèle acteur*. Dans ce modèle, un algorithme s'exprime à l'aide de N acteurs asynchrones, et son temps d'exécution parallèle se mesure par le maximum du nombre de messages traités séquentiellement pas un acteur durant l'exécution de l'algorithme. Selon ce modèle, on a défini la classe de problèmes parallèles intéressants $NC_{Acteur}$, pour lesquels, N acteurs résolvent collectivement le problème en temps parallèle $O(log^k N)$.

La problématique abordée durant notre travail de doctorat se raffine donc de la manière suivante : $NC_{PRAM} = NC_{Acteur}$ ? C'est ce que nous avons réussi à démontrer dans notre thèse et publié dans [C2,C3]. La démonstration passe entre autre par la simulation d'une PRAM sur un réseau d'acteurs, s'inspirant des protocoles mentionnés préalablement pour émuler efficacement une machine PRAM sur une machine parallèle de la première catégorie (bancs mémoire séparés des CPUs). L'émulation d'une machine PRAM sur un réseau d'acteurs devient donc assez comparable à l'implantation d'une machine PRAM sur une machine parallèle de première catégorie, avec des acteurs devant jouer le rôle de routeurs du réseau d'interconnexion. Donc, l'émulation est forcément d'un facteur logarithmique en temps plus coûteuse (en ayant égalité des 2 classes de complexité, puisque égalité des temps de calcul parallèles à des facteurs logarithmiques multiplicatifs près ...). Néanmoins, nous avons démontré qu'il n'est pas possible d'avoir une efficacité optimale : intuitivement, ceci est intrinsèque au fait qu'un acteur n'a qu'un nombre borné d'acteurs voisins, et qu'il ne peut de toute manière pas envoyer de messages en parallèle vers ces voisins. Donc, pour effectuer un routage de O(N)

requêtes parallèles d'accès en mémoire en temps $O(logN)$, il faut $O(N \cdot logN)$ acteurs. C'est à dire, qu'il faut disposer d'autant d'acteurs que l'on aurait de routeurs à degré constant, constituant le réseau de communication d'une machine parallèle de première génération.

Ce travail nous a donc permis de mieux caractériser les programmes acteurs parallèles. En effet, par la nature même du langage (concurrent, asynchrone, guidé par les messages) c'est uniquement le parallélisme dit de tâche que l'on peut exploiter. Pour autant, l'algorithmique PRAM permet d'aider à concevoir des algorithmes à passage de messages entre acteurs qui soient vraiment parallèles (donc dans la classe de complexité $NC$). S'inspirant de l'algorithmique parallèle PRAM, on exploite en fait le parallélisme provenant des données : la méthode proposée est d'associer ces données à des acteurs, et de les programmer pour que les échanges de messages initiés entre eux engendrent bien un degré de parallélisme important (en tout cas, suffisamment important pour que le temps de résolution total du problème soit dans $NC_{Acteur}$, dès lors que le problème appartient à $NC_{PRAM}$)).

Enfin, ces travaux de recherche nous ont aussi permis de catégoriser plus finement les programmes acteurs parallèles :

1. ceux pour lesquels on connait à l'avance le graphe sous-jacent représentant les échanges de messages entre acteurs ;
2. ceux, qui au contraire, n'exhibent pas a priori de graphe de communication connu avant l'exécution, car il dépend de la donnée en entrée.

L'implication de cette classification est importante : la première classe permet d'envisager d'avoir des exécutions sur machines parallèles qui conservent le même coût (c'est-à-dire temps parallèle multiplié par nombre de processeurs nécessaires) comparé à l'algorithme théorique PRAM équivalent résolvant le même problème. Ceci passe par exemple par un plongement du réseau d'acteurs sur la machine parallèle cible, qui respecte la localité (c'est-à-dire tel qu'un message envoyé par un acteur à un autre acteur soit acheminé en O(1) étapes de communication). Corroborant les résultats de travaux effectués simultanément par David Skillicorn [72], nos travaux ont donc démontré que la deuxième classe ne permet pas d'avoir une telle conservation de coût. En effet, le réseau d'acteurs doit pouvoir être capable d'exécuter efficacement le routage dynamique d'un ensemble de messages entre n'importe quels acteurs, qui ne sont pas forcément placés sur des processeurs voisins. Au mieux, on peut implanter un protocole de routage s'inspirant des protocoles d'émulations de

PRAMs sur machine parallèle à mémoire distribuée, qui comme on l'a vu plus haut génèrent un surcoût logarithmique.

## 2.2 Exploiter le parallélisme de données de manière structurée

### 2.2.1 Principes et bénéfices de l'approche structurée

Fort de cette démarche relativement théorique fondée sur l'étude de la complexité parallèle, nous est donc apparu tout l'intérêt qu'il peut y avoir à restreindre volontairement la puissance d'expression du langage parallèle :

- de sorte à n'autoriser que l'expression d'algorithmes exhibant une structure de communication prévisible
- que cette structure de communication engendrée par l'algorithme soit telle qu'elle puisse être implantée à coût constant sur n'importe quel type de machine parallèle : éventuellement grâce à un plongement optimal de la structure de communication de l'algorithme sur celle de la machine, permettant de respecter au mieux les relations de voisinage ; ou sinon, grâce à une émulation ad-hoc permettant une conservation du coût.

Concrètement, au lieu de proposer au programmeur un langage de programmation permettant d'implanter n'importe quel algorithme parallèle fondé sur l'exploitation du parallélisme de données, le langage propose seulement une palette utile, mais réduite, d'opérations permettant ensuite d'exhiber facilement et efficacement du parallélisme à l'exécution. Ces opérations sont justement celles que l'on retrouve classiquement dans les langages où le parallélisme provient de l'exploitation parallèle de structures de données, telles les listes, les tableaux :

- Les constructeurs du type `forAll, map` qui parcourent des structures de données linéaires comme des tableaux, des listes, et appliquent une opération à coût constant à chaque élément.
- Des constructeurs du type `reduction`, qui parcourent ces structures de données tout en combinant les valeurs qui y sont stockées par le biais d'opérateurs binaires, associatifs.

Notre analyse n'est pas restreinte au cas du langage acteur que nous avions considéré, dans cette approche théorique, comme un modèle basique mais illustratif du parallélisme de type MIMD. Considèrons le langage (ou bi-

bliothèque) de programmation le plus répandu, encore aujourd'hui, dans la communauté des programmeurs d'applications parallèles et réparties : MPI (Message Passing Interface). MPI permet de programmer (de manière impérative) des processus indépendants, qui se coordonnent par l'échange point-à-point de messages ou par le biais d'opérations de communication (voire aussi de calcul) collectives. Comme parfaitement argumenté en 5 points, par S. Gorlatch dans l'article [37], il y a plus d'inconvénients que d'avantages à ce que les programmeurs utilisent directement les primitives de communication point-à-point. Effectivement, cet article défend l'idée (que nous partageons) que la programmation parallèle (donc massive) utilisant les primitives d'échange de messages point à point peut être améliorée en exprimant les interactions entre processus d'une manière structurée. Concrètement, on préférera l'usage des primitives collectives de MPI (telles MPI_broadcast, MPI_reduce, MPI_scan, ...) aux primitives d'échange de messages point-à-point telles MPI_send et MPI_recv. En effet, ces primitives de communication collectives sont :

- relativement peu nombreuses et clairement spécifiées, donc simples à choisir, à utiliser, rendant le programme plus clair, plus concis,
- peuvent être sujettes à des transformations (préservant la correction) par fusion ou redécomposition, permettant de fonder une approche de programmation parallèle de haut niveau : partant d'une spécification formelle du problème à résoudre, de telles transformations permettent de systématiser la génération d'un programme parallèle correct par étapes successives,
- suffisamment expressives pour résoudre une large palette de problèmes en parallèle, offrant les principaux schémas d'interaction collective (1-N, N-1, N-N, etc),
- pour autant, implantables de manière portable et efficace (éventuellement directement au niveau du matériel, ou traduites en échanges de messages point-à-point entre processus),
- et dont les performances sont prévisibles (avant l'exécution) grâce à l'existence d'un modèle de coût qui est applicable quel que soit le type de matériel sous-jacent, paramétré par un nombre réduit de caractéristiques, telles la latence ou le débit du réseau sous-jacent.

### 2.2.2 Limitations

Le pouvoir d'expressivité du langage parallèle est donc directement dépendant des :

- structures de données à plusieurs éléments qui peuvent se voir appliquer des traitements en parallèle ; et par conséquent, quelles sont les constructions syntaxiques du langage ou fonctions de bibliothèques qui s'appliquent à ces structures (`forAll`, `map`, ..., fonctions du genre MPI_reduce qui s'applique à la structure de données qui est le buffer de réception des messages, etc.)
- des opérateurs ou fonctions pouvant s'appliquer sur les éléments regroupés dans ces structures de données.

En général, les structures de données offertes par le langage pour être gérées en parallèle sont linéaires : liste, vecteur, tableau à une dimension, ...Il nous semble que hormis leur utilité certaine pour de nombreuses applications, la raison vient de la facilité d'appliquer une division récursive en parts de taille égale pour paralléliser efficacement les traitements [4]. Il faudrait pouvoir généraliser à d'autres structures non forcément linéaires, comme arbres, graphes. Bien sûr, ceci n'est réaliste que si l'implantation d'un parcours en parallèle de telles structures de données peut se réaliser efficacement sur n'importe quelle architecture parallèle cible (permettant par conséquent de disposer d'un modèle de coût applicable en toute circonstance, via un minimum d'effort de paramétrisation).

Les opérateurs ou traitements disponibles pour s'appliquer sur la structure de données globale sont parfois imposés par le langage ou la bibliothèque. Idéalement, le programmeur devrait avoir toute la liberté de spécifier n'importe quelle fonction à appliquer à chaque élément d'une structure de données parallèle dans le cas d'un map. Et n'importe quelle fonction binaire, mais associative, pour tenir lieu d'opérateur de réduction des éléments stockés dans la structure de données dans le cas d'une réduction globale. Par exemple, dans MPI une palette d'opérateurs prédéfinis existe, permettant de calculer la somme, le max, ...La norme propose aussi la fonction MPI_Op_Create pour permettre au programmeur de définir l'opérateur de réduction qui est passé en paramètre aux opérations collectives comme MPI_reduce, MPI_allreduce, etc. Mais il revient à la charge du programmeur de faire ces créations d'opérateurs

[4] dans l'hypothèse où tous les processeurs sont de puissance équivalente, ce qui était une hypothèse réaliste dans le contexte d'alors, où les architectures cibles étaient essentiellement des machines parallèles

uniquement pour pouvoir les appliquer aux opérations collectives.

### 2.2.3 Des squelettes à parallélisme de données

Une solution permettant de combiner les avantages et palier aux limitations évoqués ci-dessus est la définition de squelettes [23]. Un squelette généralise les opérations de communication collectives, en élevant le niveau d'abstraction, et en étant complètement paramétrable. Les applications sont conçues comme des compositions ou imbrication (*nested skeleton*) de squelettes. Il existe deux familles de squelettes pour la programmation parallèle, selon qu'ils exploitent le parallélisme de données ou de tâches : les *data-parallel skeletons* (map, reduce, scan, etc) et les *task-parallel skeletons* (pipeline, maitre-esclave, ferme, etc). La première catégorie est celle qui nous intéresse tout particulièrement dans ce chapitre (l'autre catégorie nous intéressant dans le cadre de certains des travaux prospectifs évoqués en section 5.1).

Un squelette se rapproche donc d'un schéma de conception, prédéfini dans le langage ou la bibliothèque. Le point clé est que son implantation est complètement masquée à l'utilisateur, et prend en charge la gestion de la répartition et de la synchronisation. Une telle approche permet à l'utilisateur de bénéficier de manière transparente d'optimisations (statiques et dynamiques) lors de l'implantation des squelettes, en fonction de l'architecture cible et des conditions d'exécution (par exemple, prise en compte de la charge induite par d'autres processus concurrents sur l'architecture cible). Ainsi, une approche par squelette a en plus le mérite de favoriser la réutilisation de code, en particulier, du code dit non fonctionnel.

De nombreuses implantations du concept de squelette ont été réalisées dans des langages de programmation fonctionnels, puisque on peut voir un lien naturel entre un squelette et une fonction d'ordre supérieur. Ceci permet en particulier de résoudre la seconde des limitations évoquées, concernant l'expression de n'importe quel traitement ou opérateur binaire à appliquer aux éléments du squelette.

## 2.3 Types de données catégoriques parallèles

Le travail réalisé durant une partie de notre stage post-doctoral, dans l'équipe de David Skillicorn (voir article [J1], repris en [Section 2.1.1, page 95]), a contribué à lever la premiere des limitations évoquées, à savoir la limita-

tion des squelettes à parallélisme de données aux seules structures de données linéraires, telles les listes.

### 2.3.1 Fondement d'une méthode de programmation parallèle basée sur des types de données catégoriques

Pour cela, la base est la théorie des catégories. Le principe est de généraliser le formalisme Bird-Merteens, qui définit dans cette théorie, le type liste, et de l'appliquer à d'autres structures de données complexes, non forcément linéraires [73].

L'idée intéressante dans la théorie appliquée aux listes, et qui fonde une approche de programmation parallèle est la suivante : une fonction sur une liste est un homomorphisme dès lors qu'elle peut s'exprimer comme la composition d'opérations de type map et reduce. Dans la théorie, les seules fonctions autorisées sont les homomophismes. Ici, une opération *reduce* est prise dans son sens le plus général, c'est-à-dire où le traitement sur la structure de données donne lieu à une structure de données différente (par exemple une liste d'entiers vers un entier, ou une liste d'entiers vers une liste de listes d'entiers, etc). Sachant que map et reduce sur des structures linéraires s'implantent efficacement et au même coût quel que soit le type sous jacent d'architecture parallèle [72], on a là une méthode de programmation parallèle à la fois générique et portable, et même équationnelle (les programmes peuvent se raffiner, à l'aide d'équations définies dans la théorie [76]).

La théorie des catégories permet de généraliser cette approche centrée sur les listes à n'importe quel type de données catégoriques [73]. En se référant à la terminologie orientée objet, les seules méthodes publiques disponibles sur ces types sont des homomorphismes (nommés en fait *catamorphismes*). L'intérêt, à nouveau, est que tout le parallélisme inhérent à l'application d'un catamorphisme provient d'un parcours de la structure de données reflétant sa construction récursive, et les statégies (algorithmes, compilations, etc) pour exploiter voire optimiser de tels parcours peuvent rester entièrement transparentes au programmeur. Un point délicat dans l'exploitation du parallélisme reste cependant l'enchainement, ou la composition, de plusieurs catamorphismes. En effet, entre l'application de deux catamorphismes, il peut être nécessaire de répartir différemment les données sur les différents processeurs. La théorie n'apporte pas de solution en soi. Par contre, en remplaçant des catamorphismes par d'autres, les mouvements de données nécessaires au final

peuvent être moindres.

### 2.3.2 Nécessité de modèles de coût

Cependant, pour que cette approche soit le socle d'une méthode de programmation parallèle exploitant efficacement le parallélisme de données, il est primordial de pouvoir définir un modèle de coût (voir en particulier la section 3.2 de l'article, [Section 2.1.1, page 95]) : pour chaque type de structure de données ou squelette, et pour chaque catamorphisme, on aimerait définir un coût d'implantation du catamorphisme sur une architecture donnée (en fonction de la taille de la donnée, nombre de processeurs et temps parallèle). On veut aussi pouvoir s'assurer qu'il existe pour chaque architecture visée, une implantation qui exhibe le même coût. Ceci est requis si l'on veut avoir une méthode de programmation parallèle indépendante de l'architecture cible, donc une méthode portable. Disposant d'un modèle de coût, l'intérêt est qu'un programme parallèle s'obtient par transformations successives, partant d'une spécification initiale d'une fonction à appliquer sur le type de données, et aboutissant à une formulation la plus optimisée possible en fonction de l'architecture ciblée. Notamment, c'est grâce à la connaissance des coûts associés à chaque membre gauche et droite de chaque équation disponible dans la théorie, que le raffinement de la spécification initiale peut conduire à une implantation efficace, voire optimale [76, 73].

> Une telle méthodologie de programmation était déjà largement explorée concernant les listes, qui sont des structures linéaires (donc a priori, sur lesquelles map et reduce sont faciles à paralléliser et pour lesquelles des modèles de coût existent). Mais, la méthodologie pouvait-elle s'appliquer à d'autres types de données structurées ?

### 2.3.3 Généralisation à des structures de données non linéaires

Durant une partie de notre séjour post-doctoral, nous nous sommes focalisés sur le cas de la structure de données en arbre binaire.

Les arbres binaires, et plus généralement les arbres de Rose (où chaque nœud de l'arbre peut avoir un nombre quelconque de fils), sont à la base de nombreux algorithmes intéressants : effectivement, tout traitement sur

un document structuré à l'aide de paranthèses comme le cas de programmes fonctionels, ou de balises comme dans le cas de documents XML, peut potentiellement s'exprimer comme un traitement sur un arbre [75]. On distingue plusieurs classes d'opérations qui s'expriment comme des (compositions d') homomorphismes : map, réduction des valeurs stockées dans l'arbre, accumulation (sorte de somme préfixe) des feuilles vers la racine, ou inversement. Cela permet d'exprimer tout traitement de comptabilisation ou recherche de motifs dans un document (par exemple une accumulation de la racine vers les feuilles peut permettre d'obtenir la liste de toutes les références d'une étiquette donnée dans un document).

Plus précisément, le problème qui nous a occupé était d'étudier l'existence d'algorithmes parallèles pour machines parallèles sans mémoire physique commune, permettant d'effectuer la réduction des valeurs stockées dans l'arbre (contraction d'arbre). Sachant que le problème est dans $NC_{PRAM}$, l'objectif était d'exhiber un algorithme de réduction des valeurs stockées dans les N feuilles d'un arbre binaire, qui soit idéalement, en temps parallèle $O(logN)$ sur au plus N processeurs sans mémoire physique commune. Par chance, un algorithme ayant ces performances avait été tout juste défini [57] pour une topologie d'interconnexion en forme d'hypercube. Le fait d'avoir identifié l'intérêt de cet algorithme est certes moins prestigieux que de l'avoir conçu, mais cela a eu un impact considérable sur la suite de l'élaboration de la méthode générale de dérivation d'algorithmes parallèles sur le type de données catégorique arbre [74, 55].

En effet, toute la force de cet algorithme est de réussir à avoir un temps parallèle qui ne dépend pas de la profondeur de l'arbre, mais du logarithme du nombre de ses feuilles, sans pour autant requérir l'hypothèse d'une mémoire commune. Ainsi, cet algorithme permet d'exécuter efficacement n'importe quel catamorphisme engendrant des communications dans l'arbre selon les relations de parenté, (donc pas seulement la reduction mais aussi les accumulations). C'est d'autant plus remarquable que l'arbre a une structure typiquement irrégulière, donc difficile à partitionner en parts de taille égale [56], et encore plus difficile à plonger sur la topologie physique en respectant au mieux les relations de voisinage entre processeurs [R3].

### 2.3.4 Bilan

Fort de ce type de résultats, une approche fondée sur des squelettes exploitant en parallèle des structures de données de grande taille régulières

ou irrégulières telles des arbres devient possible. L'objectif est de faciliter la tâche du programmeur en le déchargeant de la gestion du parallélisme. Le point critique dans cette gestion est celui du choix de la granularité du parallélisme, car de ce choix découlent les schémas de communication et de synchronisation à réaliser à l'exécution. Grâce à l'existence de modèles de coût, il devient possible d'envisager des approches de programmation par transformation, qui dérivent automatiquement du code efficace, y compris pour des supports de calcul réparti sans mémoire commune [73]. Mais, il faut bien être conscient que ceci n'est possible que parce que les problèmes à résoudre sont reportés du niveau applicatif sur le niveau sous-jacent. En effet, c'est au système en charge de dériver une implémentation réaliste à partir des spécifications de l'application que revient le choix de la granularité, la désignation des synchronisations et des transferts de données à réaliser entre l'application successive d'opérations globales sur les structures de données [77]. Par ailleurs, le choix de telle ou telle fonction paramétrant un squelette, avec comme objectif de minimiser les mouvements de données est loin d'être simple (voir par exemple [55]). Le bénéfice escompté de tels systèmes est que les décisions de mise en œuvre sont prises le plus tard possible, non plus au moment de la spécification de l'application, mais, au moment de sa compilation en tenant compte du contexte dans lequelle elle devra s'exécuter.

Au moment où ces travaux étaient réalisés, les machines cibles étaient des machines parallèles de la première ou de la seconde catégorie (voir section 2.1), où les performances de calcul et de communication sont homogènes et où toutes les ressources utilisables sont connues à l'avance. Avec l'avènement des systèmes de type metacomputing comme les grilles de calcul où la disponibilité des ressources est variable, et les supports de calcul et de communication ont des performances hétérogènes, ces hypothèses ne sont plus vérifiées. Des travaux plus récents ont du ainsi être menés.

Un exemple intégré d'une telle démarche est présenté dans [45], proposant le concept d'*Intensional HPC*. D'une même application parallèle décrite comme une imbrication hiérarchique de cellules parallèles (cellules soit élémentaires, soit constituées d'aggrégats), toutes les implantations parallèles possibles doivent pouvoir être extraites (à une extrémité du spectre, l'implantation est entièrement séquentielle, à l'autre, toute cellule est supportée par un processus parallèle). Un autre exemple est décrit dans l'article [8] : il propose de choisir les ressources de calcul sur la grille grâce à la connaissance préliminaire des niveaux de performances de ces ressources dans l'exécution complète – invocation, composition, renvoi des résultats – des squelettes dis-

ponibles dans le langage. On peut aussi recourir à des stragégies qui sont effectives à l'exécution : le compilateur engendre un ensemble de tâches pour implémenter le squelette ; ce n'est qu'à l'exécution qu'on affecte (ou réaffecte) dynamiquement ces tâches sur les processeurs disponibles, en fonction de leur charge courante [84], [51].

Dans ce contexte de ressources de calcul hétérogènes, il semble donc nécessaire de reposer sur une approche la plus portable et paramétrable possible pour implanter de tels squelettes, ou plus généralement toute structure permettant de gérer proprement le parallélisme. Baser un tel système autour de Java et RMI, particulièrement adéquat pour la programmation répartie en environnement hétérogène est un point de départ raisonnable [9], [25] que nous partageons. Le choix d'un mode de programmation ayant de bonnes qualités en terme de génie du logiciel, tel que l'approche orientée objets est par ailleurs utile. Cependant, pour qu'un tel système soit efficace, Java RMI seul est loin d'être satisfaisant (par exemple, trop contraignant car les appels de méthode, qui se traduisent en envois de messages, sont synchrones). La suite du document présente justement des extensions à la programmation répartie en Java. Nous pensons que de telles extensions constituent des briques pertinentes pour le support de paradigmes de haut niveau d'abstraction tels les squelettes afin de faciliter la programmation d'applications en vue de leur exécution sur des grilles de calcul.

## 2.4 Conclusion

Au milieu des années 1990, où les supports de calcul parallèles n'étaient plus majoritairement des machines parallèles, mais devenaient de plus en plus des stations de travail ou PCs reliés autour d'un réseau local, le succès des bibliothèques de programmation parallèle et répartie par passage de messages, portables, comme PVM et MPI ne faiblissait pas. Malgré ces efforts de portabilité, la transparence pour le programmeur concernant la gestion du parallélisme et de la répartition n'était malheureusement pas au rendez-vous. Comme nous l'avons résumé dans ce chapitre, nous avions déjà pris conscience de l'importance de disposer d'un système opératoire portable (autour de la notion d'acteur [W1]), jouant le rôle d'intermédiaire entre le programme et l'architecture cible, facilitant la tâche du programmeur, sans pour autant empêcher des optimisations ultérieures.

Nous avons alors eu l'opportunité de rejoindre le groupe formé autour de

Denis Caromel, s'intéressant à la gestion de la concurrence et de la répartition dans un cadre de programmation orientée objet ; et de participer ainsi au montage du projet SLOOP *Simulation, Langages à Objets et Parallélisme*, commun entre l'I3S, l'UNSA et l'INRIA, sous la direction de Jean-Claude Bermond. Notre objectif était relativement concret : définir une architecture permettant de programmer et exécuter de manière concurrente et parallèle sur réseau de stations de travail, des applications difficiles à paralléliser, car à parallélisme de tâche irrégulier, et de grande taille, telles des simulations à événements discrets [W3,J2]. Le besoin d'un système opératoire portable, efficace, pour supporter de telles extensions aux langages orientés objet, comme C++ puis Java, a donc constitué notre nouvel objectif de recherche.

En fin de compte, nous verrons que les résultats de nos recherches initiées dans ce nouveau contexte se rattachent en quelque sorte à notre problématique initiale d'expression structurée puis d'exploitation du parallélisme et de la répartition. Vue la complexité de programmation et de mise en œuvre d'applications sur grilles de calcul, une approche par squelettes, transparente pour le programmeur est particulièrement attirante. Cependant, ce n'est que grâce aux travaux les plus récents, introduisant la notion de *squelettes de seconde génération* [24], qui plus est embarqués ou combinés à une approche par composants logiciels [66], qu'une telle approche nous semble être maintenant suffisamment mûre pour résoudre des applications réalistes. Nous y reviendrons dans le dernier chapitre.

# Chapitre 3

# Structuration par Objets Actifs Mobiles

Ce chapitre résume de la façon la plus synthétique possible la démarche et les résultats obtenus autour de la définition d'environnements de programmation par objets répartis, avec comme perspective, l'ajout d'une couche à composants logiciels. Nous nous focalisons sur les applications parallèles et réparties en vue de les exécuter sur les types de support largement répandus aujourd'hui, soit, des réseaux de station de travail aux grilles de calcul, en passant par les machines multiprocesseurs et les grappes de PCs à haute performance. Tous ces efforts ont été menés en collaboration trés étroite avec Denis Caromel, au sein du projet SLOOP puis du projet OASIS depuis 1999, autour des doctorants qui se sont succédés durant ces années, et dont les travaux constituent évidemment des contributions importantes : Yves Roudier, Nathalie Furmento, David Sagnol, Julien Vayssière, Fabrice Huet, Laurent Baduel, Arnaud Contes, Christian Delbé et Mario Leyton ; sans oublier la collaboration de Lionel Mestre, Romain Quilici, Matthieu Morel, Virginie Legrand, tous ingénieurs de recherche et de développement qui se sont succédés au fil des années, et sans qui notre ambition collective à proposer des solutions qui soient utilisables au delà du domaine académique ne pourrait être fondée.

Les contributions présentées dans ce chapitre ont été intégrées successivement au sein de deux environnements de programmation parallèle et répartie, tous deux se présentant sous la forme

1. d'une bibliothèque, n'impliquant aucune modification du langage, quasi-transparente d'utilisation pour le programmeur, et ce grâce à des tech-

niques de réflexion et l'usage d'un protocole à méta objets,

2. couplée à un support d'exécution permettant concrètement de cibler des environnements répartis, avec un objectif récurrent et fort : la portabilité du niveau applicatif via une bonne isolation vis-à-vis du support cible.

Ces deux environnements sont :
- C++// : extension de C++ pour la répartition,
- ProActive, extension de Java pour le parallélisme et la répartition, constituant depuis Avril 2002, un des projets d'ObjectWeb, *OpenSource Middleware consortium* : `www.objectweb.org/proactive`.

## 3.1 Caractéristiques du niveau applicatif

Une brève introduction présente les bases du modèle de programmation répartie que nous prônons ; ce modèle rend aisée la gestion de la répartition pour l'utilisateur final ainsi que la réutilisation de code séquentiel existant. Puis, nous présentons successivement les trois principales extensions au modèle que nous avons contribué à définir. Ces extensions visent autant la gestion de la répartition que du parallélisme.

### 3.1.1 Modèle de programmation de base

Les principes de base, appliqués aussi bien dans C++// que dans ProActive, se résument ainsi (voir figure 3.1) :
- les objets peuvent être rendus asynchrones et répartis, par héritage : on parle alors d'objet actif. Un objet actif peut être considéré comme une sorte d'extension d'un acteur servant des messages séquentiellement[1] avec en plus, les possibilités de structuration, de réutilisation de code que l'approche orientée objet apporte
- tous les objets ne sont pas forcément actifs, et il n'y a pas de partage entre objets actifs (les objets passifs passés en paramètre le sont par copie), ce qui permet de moduler la granularité des tâches et n'engendre pas de trop fortes contraintes de co-localisations lors de la répartition

[1] on parle alors d'acteur au comportement *sérialisé* ou *mutable*, par opposition aux acteurs non sérialisés [5] qui ne changent pas d'état, et peuvent donc traiter les messages d'une manière concurrente

– tout appel de méthode vers un objet actif a la même syntaxe que vers un objet passif, mais la sémantique diffère. L'appel est automatiquement transformé en un appel asynchrone avec rendez-vous (le rendez-vous permettant de garantir que l'appel est bien parvenu à l'objet cible)
– un principe qui permet grandement de faciliter la réutilisation de code existant provient du fait que les messages véhiculant les requêtes d'exécution de méthode sont mis dans une queue de requêtes ; le service de ces requêtes est FIFO par défaut, et, crucial pour la réutilisation de code, l'objet actif est un processus qui sert les méthodes séquentiellement (on n'a donc aucune concurrence à gérer dans l'exécution des méthodes)
– les réponses aux méthodes invoquées sur des objets actifs sont automatiquement transformées en promesses de réponses (*futur*), et l'appelant peut ainsi poursuivre son exécution sans devoir attendre la fin de l'exécution de la méthode. Par contre, une attente par nécessité de l'appelant est effectuée de manière totalement transparente, dès lors qu'il tente d'utiliser la réponse et que celle-ci n'est pas disponible.

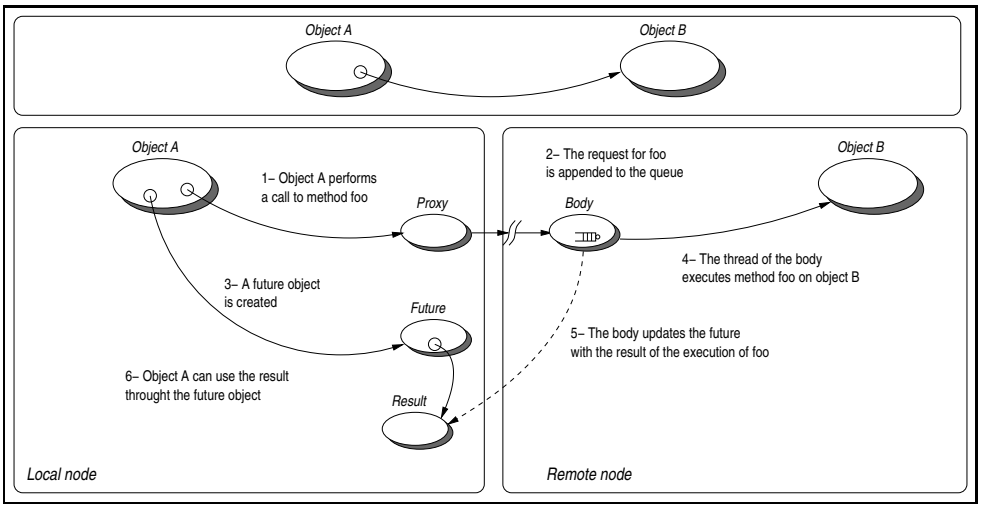



FIG. 3.1 – Modèle d'objet actif

Tous ces principes sont détaillés, ainsi que certaines optimisations de mise en œuvre, pour C++// et respectivement ProActive, dans [Section 2.2.1, page 111+2] et respectivement [Section 2.2.2, page 139+2].

### 3.1.2 Recouvrement calcul et communication

#### Exposé de la problématique

Dans le contexte d'appels de méthode à distance, notre modèle de base asynchrone rend possible un véritable parallélisme dans l'exécution des opérations (au sens large, c'est-à-dire communications ou exécution de méthode) se déroulant côté appelant et côté appelé. Ainsi dans le modèle de base, pour une paire donnée d'objets actifs, on peut réussir à masquer les coûts de communication correspondant à l'acheminement à distance d'une requête, à partir de la seconde requête envoyée vers le même objet actif.

Mais on peut essayer d'aller plus loin dans l'exploitation de recouvrements calcul - communication : on peut essayer de masquer en partie les coûts de communications engendrées par *un* appel de méthode distant, car ces coûts peuvent être importants si les paramètres d'appel sont volumineux. Cependant, dans le modèle de base, exprimer un tel potentiel de recouvrement requiert une structuration explicite au niveau du code utilisateur : dans le cas précis où le traitement distant est long, car devant traiter un volume important de données, le programmeur est obligé de décomposer ce traitement en plusieurs sous-traitements, chacun d'eux s'occupant d'un sous-ensemble des données[2]. Une fois cet effort de programmation réalisé, il peut alors résulter un recouvrement des communications par les calculs exécutés à distance ; ce recouvrement est obtenu par l'enchainement des appels distants et asychrones de ces différents sous-traitements.

Evidemment, un tel découpage en sous-traitements nuit à la clarté du code de l'utilisateur, puisque il est explicite. Par ailleurs, pour engendrer un gain de performances, la programmation de ce découpage doit d'une certaine manière être guidée par les performances de communication sous-jacentes. Du coup, un tel découpage explicite peut n'engendrer aucun gain si l'on change d'environnement d'exécution, voire même nuire aux performances qui seraient obtenues sans tenter d'exploiter du recouvrement calcul/communication.

Résoudre un tel problème de découpage, de manière à la fois transparente vis-à-vis du code de l'utilisateur, et optimale, c'est-à-dire, en proposant la meilleure décomposition étant donné l'environnement cible, n'est pas possible, du moins dans notre contexte. Par contre, nous avons contribué à

[2]Une restriction existe évidemment : le gros traitement sur le volume complet des données doit être décomposable en sous-traitements indépendants, et ce sur des volumes de données moins importants

cette problématique de recherche, en proposant un compromis entre transparence et optimalité, dans le cadre des langages à objets répartis. Il s'agit d'un mécanisme original permettant du recouvrement calcul et communication, présenté très brièvement ci-dessous, et qui a fait l'objet spécifique des publications [C4,C7,J3].

### Application de la notion de *futur* aux paramètres d'appel

Les principes et résultats se trouvent décrits plus amplement en [Section 2.2.1, page 111, section 6]. L'ensemble est détaillé dans la thèse de doctorat de Nathalie Furmento.

On réutilise la notion d'attente par nécessité qu'offre le mécanisme de futur, qui s'applique d'habitude aux réponses des méthodes invoquées sur des objets actifs. Lorsque le programmeur désire bénéficier d'une optimisation issue d'un recouvrement calcul - communication, il peut appliquer ce mécanisme de futur aux paramètres d'un appel de méthode vers un objet actif distant : comme ces paramètres arriveront un peu plus tard que la requête d'exécution de la méthode, on les qualifie de retardataires (*later*). L'envoi d'un paramètre, ou d'une portion d'un paramètre, peut potentiellement être recouvert par l'exécution – partielle – de la méthode invoquée sur l'objet actif, dès lors que l'on ne se sert pas encore de ce(s) – portion(s) de – paramètre(s). Le mécanisme de futur assure de manière transparente qu'une attente par nécessité aura lieu dès lors que le traitement aura besoin des paramètres manquants.

> Au prix d'un effort minimum de la part du programmeur, qui n'a pas vraiment besoin d'expliciter un découpage des traitements et des paramètres concernés, cette technique offre un potentiel de recouvrement calcul - communication.

Les résultats expérimentaux que nous avons menés confirment ce potentiel ([Section 2.2.1, page 111, figure 14]).

Ce mécanisme a été développé dans le contexte de C++//. Il n'a pas été introduit dans ProActive, bien que le principe pourrait s'appliquer. Mais depuis, ProActive propose un mécanisme dit de *continuation automatique*, complémentant, voire généralisant en quelque sorte, cette notion de retardataire : une promesse de réponse peut être passée en paramètre d'un nouvel appel de méthode sur un objet actif (même sans aucune annotation préliminaire dans la signature de méthode). Bien sûr, le mécanisme d'attente par nécessité

est activé de manière transparente. De plus, la mise à jour de toutes les copies d'une promesse de réponse est effectuée par le système, de manière transparente, et peut l'être selon différentes stratégies [20].

### 3.1.3 Mobilité

#### Intérêt pour la mobilité

Code et activités mobiles (objets actifs, acteurs, agents, etc) apparaissent comme une solution prometteuse permettant la construction d'applications flexibles et dynamiquement adaptables aux contraintes de l'environnement d'exécution, dans le cas où ce dernier est réparti [12]. Dans le cadre du calcul réparti haute performance, notamment sur grille de calcul, la possibilité de déplacer une activité d'un site à l'autre présente de nombreux intérêts, dont par exemple :

- réagir à une indisponibilité prévue de la machine hôte, en déplaçant l'activité pour qu'elle se poursuive ailleurs
- mieux répartir la charge de calcul des différents sites, et ce dynamiquement.

La mobilité d'activité a également un intérêt pour :

- appréhender des applications structurées autour du suivi d'un itinéraire ; typiquement, une application de commerce électronique qui en vue de réaliser des achats doit visiter plusieurs sites marchands, éventuellement en parallèle pour optimiser le parcours de l'itinéraire
- proposer des applications adaptées au fait que l'utilisateur est mobile ; typiquement, une activité permettant l'interaction de l'utilisateur avec l'application doit pouvoir accompagner cet utilisateur sans aucune incidence sur l'application en cours, si l'utilisateur passe d'un support d'exécution à un autre[3]. Un scénario intéressant, et pas si futuriste que cela, est celui où l'utilisateur quitte son PC professionnel, car il part en réunion avec son ordinateur portable, passe ensuite dans sa voiture équipée d'un ordinateur de bord performant, rentre à son domicile où il dispose bien sûr d'un ordinateur, passe la soirée en dehors de chez

[3]une hypothèse importante est néanmoins que chacun de ces types de supports soit accessible depuis Internet. Dans le cas contraire, il est prévu d'explorer plus en profondeur la possibilité d'avoir un mode de communication déconnecté, mémorisant les requêtes ou les réponses tant que la connexion réseau n'est pas disponible, plutôt que de basculer en erreur

lui, auquel cas il emmène avec lui son assistant personnel. Sa mission est d'être constamment connecté à l'application qu'il doit surveiller, celle-ci devant être disponible 24/24/7/7.

On remarque un point commun entre tous ces cas de figure d'utilisation d'activités mobiles participant à une application parallèle et répartie : les entités mobiles doivent pouvoir continuer à être atteignables quel que soit leur lieu d'exécution. Nous nous sommes attaqués à cette problématique : faire en sorte que mobilité et communication distante puissent cohabiter, de manière flexible et performante.

### Une bibliothèque de gestion de la mobilité en présence de communications distantes et asynchrones

Les articles [C5,J4] présentent spécifiquement la solution que nous proposons pour permettre de programmer des objets actifs mobiles dans la plateforme ProActive, et se trouve résumée en [Section 2.2.2, page 139+3] et [Section 2.4.1, page 221+2]. L'ensemble est détaillé dans la thèse de doctorat de Fabrice Huet.

La bibliothèque proposée permet uniquement de la migration faible pour les objets actifs : ceci signifie qu'un objet actif ne migre qu'à des points bien particuliers de son exécution. Plus précisément, seulement lorsqu'il atteint un point où on est en mesure de sérialiser son état, ce qui dans le modèle à objets actifs correspond aux points situés entre les services de requêtes. Une requête de migration a exactement la même forme qu'une requête correspondant à une demande d'exécution de méthode et se trouve placée dans la queue des requêtes de l'objet actif. Un mécanisme d'itinéraire permet d'exprimer aisément une succession de migrations à effectuer, et de programmer, simplement, certaines actions à réaliser au départ ou à l'arrivée sur un site de l'itinéraire.

La principale difficulté est de s'assurer que l'envoi d'une requête[4] vers un objet actif mobile respecte bien l'ordre FIFO point-à-point entre toute paire d'objets actifs, constituant un fondement incontournable du modèle de base, et qui est assuré par le mécanisme de rendez-vous.

Notre solution consiste tout simplement à prolonger le rendez-vous jusqu'au moment où le message réussit à atteindre l'objet mobile cible. Nous avons

[4] pas forcément d'une réponse[20]

étudié plus précisément deux stratégies, classiques dans le domaine, afin de localiser un objet mobile [C5,J4] : une approche par répéteur (*forwarder*), une approche par serveur de localisation, et enfin, contribution originale du travail de Fabrice Huet, la possibilité d'utiliser dynamiquement l'une ou l'autre stratégie, en fonction de paramètres propres à l'exécution, comme le délai entre deux migrations ou le délai pour mettre à jour un serveur de localisation [19]. Quelle que soit la stratégie retenue, notre approche garantit que dès la fin du rendez-vous, l'émetteur est en mesure d'*actualiser* l'adresse qu'il possédait pour joindre le destinataire.

Des travaux en cours exploitent cette approche de mobilité pour réaliser de l'équilibrage de charge dynamique [16]. Un autre contexte d'utilisation est celui d'un système de calcul de type pair-à-pair conçu au-dessus de la plateforme ProActive [18] ; la mobilité sert ici pour s'adapter au fait qu'une ressource de calcul peut devenir indisponible car son propriétaire ne désire plus qu'elle fasse momentanément ou définitivement partie du système : on peut donc envisager de migrer tous les objets actifs en cours d'exécution vers une autre ressource de calcul. D'autres applications de la mobilité sont par exemple l'administration système et réseau. Ceci fait l'objet d'une partie du chapitre suivant.

### 3.1.4 Groupes typés d'objets

Schématiquement, l'exploitation du parallélisme issu des données résulte en un ensemble de traitements similaires, chacun portant sur une portion de ces données. Ce paradigme est à la base du mode de programmation parallèle SPMD (*Single Program Multiple Data*), fondement de la bibliothèque MPI via laquelle les programmes parallèles sont organisés autour de processus à peu près équivalents (hormis un d'entre eux, qui joue en général un rôle supplémentaire de chef d'orchestre). Dans le contexte de la programmation orientée objet répartie le mécanisme de base est l'objet actif, qui se compare à la notion de processus MPI. Pour que l'approche orientée objet répartie puisse prétendre servir d'outil de programmation parallèle, il manquait clairement un niveau de structuration : la notion de groupes d'objets participant au même traitement parallèle, donc d'un rassemblement d'objets offrant la même fonctionnalité.

> Notre contribution dans ce domaine est de prolonger l'idée d'un objet typé dont les méthodes sont invoquables à distance de

> manière asynchrone, en un *groupe typé* d'objets – de type compatible, sur lequel l'invocation d'une méthode est propagée en parallèle à tous les membres du groupe. En complément de ce mécanisme, nous proposons un certain nombre de facilités que l'on s'attend à trouver dans une bibliothèque de programmation SPMD : organisation des processus selon des topologies virtuelles, barrières de synchronisation.

Les articles [C10,C16] présentent le mécanisme des groupes d'objets actifs, et l'article [C17] son extension pour la programmation SPMD dans un cadre objet (nommée OO-SPMD par la suite). Ils sont résumés en [Section 2.2.2, page 139, section 2.4] et [Section 2.2.3, page 171]. L'ensemble est détaillé dans la thèse de doctorat de Laurent Baduel.

Ce mécanisme a été démontré sur une application parallèle non triviale de simulation de propagation d'ondes électromagnétiques, dont il existait une version écrite en Fortran et MPI ; elle a donc été redéveloppée en utilisant ProActive et en utilisant des groupes d'objets actifs [C13]. Le mécanisme des groupes peut-être utile pour implémenter de manière optimisée des interfaces collectives de composants logiciels, sur lesquelles les invocations sont multiples et peuvent donc être optimisées si implémentées grâce à des groupes d'objets actifs. Une telle utilisation est explicitée notamment dans les articles [C12,C14,C15,C18]. Finalement, les groupes sont un moyen d'optimisation en présence d'un nombre important d'objets actifs et de ce fait, devrait aider au passage à l'échelle. C'est ce que nous explorons spécifiquement dans le cas de l'administration distante d'un grand nombre de passerelles OSGi et fait l'objet d'une partie du chapitre suivant.

## 3.2 Caractéristiques du niveau exécutif

**Problématique** La démocratisation des applications parallèles et réparties provient notamment du fait d'un accès plus facile, moins honéreux à des supports de calcul parallèles formés à base de matériel du commerce (réseaux locaux de stations de travail, grappes de PCs, etc). Tous ces parcs ne sont pas homogènes, que ce soit en terme d'architecture ou de système d'exploitation. Cette hétérogénéité est encore excacerbée dans le cas de grilles de calcul où les décisions d'acquisition de matériel et de logiciels sont par nature pas du tout coordonnées. Un environnement de programmation et d'exécution parallèle et répartie se doit donc de considérer sérieusement le problème de

la *portabilité* sur la plus large gamme possible de matériels et de systèmes. Le choix de Java comme langage support de l'environnement est déjà un élément clé pour atteindre cet objectif.

> Notre contribution dans ce domaine est de fonder la plateforme sur un système à l'exécution qui soit *ouvert*, et qui sert de couche de portabilité se déclinant en deux axes principaux :
> – indépendance vis-à-vis du protocole de transport des messages
> – indépendance vis-à-vis du protocole de création des supports d'exécution (nécessitant le cas échéant l'approvisionnement distant de fichiers de code et de données).

Le caractère *ouvert* de notre proposition mérite d'être souligné. En effet, nombreux sont les environnements de programmation et d'exécution qui sont très vite devenus obsolètes, car étroitement liés aux caractéristiques des supports cibles du moment. Nous pensons que la seule solution face à de telles évolutions, rapides, incontournables et logiques dans le domaine de l'informatique, consiste à proposer une couche de portabilité qui soit elle-même capable d'évoluer[5] en fonction des couches basses, sans pour autant remettre en cause le modèle offert aux couches hautes, point clé pour que ces dernières soient justement portables. La suite de cette section 3.2 reprend successivement les deux axes évoqués. Notre vision, que la plateforme ProActive implémente, se trouve résumée en [Section 2.2.2, page 139, sections 2.5 et 4], et comparée de manière synthétique à d'autres approches pertinentes pour programmer les grilles de calcul, en Section 2.2.4, page 181.

### 3.2.1 Indépendance vis-à-vis du transport des messages

Dans le contexte de la thèse de doctorat de Nathalie Furmento, qui participait de la conception et l'implémentation du système C++//, notre rôle était d'apporter une solution de portabilité face à la multitude des supports systèmes pour les activités de C++// et leurs interactions. Ceci s'est concrétisé par une bibliothèque nommée Schooner, présentée dans [C3,W4,J2]. Schooner consiste en une encapsulation orientée objet de supports d'exécution. Le fait que cette encapsulation soit orientée objet permet l'extensibilité ; la liste, mais par définition, non limititative, des supports d'exécution envisagés était la suivante :

[5] donc ouverte, extensible, paramétrable

- concernant les supports d'activités : processus lourds Unix, processus légers de niveau système ou de niveau utilisateur
- concernant les protocoles de transports de messages associés à ces différents supports d'exécution : PVM, MPI, RPC entre processus lourds ou légers comme PM2 [62] ou Nexus [31],

la majorité de ces systèmes se présentant déjà comme une solution de portabilité vis-à-vis des couches sous-jacentes. Notre effort a porté sur la définition d'une couche intermédiaire qui réussit à unifier l'approche par échange de messages point-à-point, et l'approche par invocation de service distant qui se rapproche du concept de message actif [83], pour permettre par la suite le transport de requêtes et de réponses de C++// ou du noyau de simulation parallèle à événements discrets, Prosit [J2].

Le choix du langage Java dans la conception de ProActive a dans un premier temps largement diminué le spectre des supports cibles envisagés : le choix de threads Java couplé au système RMI standard pour le transport des requêtes et réponses entre objets actifs était évident. Néanmoins, il nous est vite apparu que ces choix pouvaient être revus : il existe des implémentations de Java RMI performantes, Java RMI utilise des ports de communication IP dont l'usage peut être impossible à cause de pares-feux, ...Effectivement, le remplacement de RMI standard par une autre implémentation, Ibis [64] a permis de belles performances dans l'exécution de l'application de simulation parallèle d'ondes électromagnétiques, Jem3D [39][C18]. Par ailleurs, l'utilisation du protocole HTTP plutôt que RMI, ou de RMI sur SSH, sont des bases intéressantes : les supports d'exécution (JVM) ProActive et les objets actifs qu'ils hébergent peuvent être déployés de manière hiérarchique (travail en cours dans l'équipe), y compris vers des sites où seuls les ports HTTP ou SSH sont ouverts. Bien évidemment, le choix de tel ou tel protocole de transport doit pouvoir être fait dynamiquement : a priori, le choix est fait par l'émetteur, en fonction de ce que le destinataire annonce comme être le protocole préféré pour être joint, autorisant que cette caractéristique soit elle même évolutive dans le temps.

### 3.2.2 Indépendance vis-à-vis du lieu d'exécution

Un point difficile et critique dans la gestion du parallélisme et de la répartition a toujours été le placement, statique ou dynamique, des différentes activités sur les ressources disponibles. Sans pour autant chercher à résoudre le difficile problème consistant à optimiser le placement en fonction des be-

soins en terme de ressources de calcul et de comunication des différentes activités de l'application, un point que nous pensons être critique, et que nous avons abordé est le suivant :

> Comment orchestrer et contrôler l'acquisition des ressources de calcul, pour qu'elles soient ensuite mises à disposition de l'application, afin d'y placer les différentes activités ?

Dans MPI par exemple, le lancement de l'application parallèle est effectué grâce à une commande externe `mpirun` qui a comme objet principal de sélectionner des ressources de calcul, pour lancer l'ensemble des processus parallèles. C'est au moment de ce lancement que chaque processus se voit attribuer un rang, rang utilisé le cas échéant au sein du programme pour particulariser le rôle d'un processus. Comme on peut le constater, il y a une très forte isolation entre le programme et son environnement d'exécution, le seul lien existant étant les rangs des processus. Le programme lui même n'exerce aucun contrôle sur son environnement d'exécution.

Considérant le calcul parallèle à grande échelle sur grille, il se peut que tous les processus participant à l'application n'aient pas à être démarrés en même temps, initialement. De même, certaines applications se présentent comme des services, dont la vocation est d'être utilisables par des applications tierces. En résumé, on est en présence d'un ensemble de codes couplés d'une manière plus ou moins lâche, pas forcément déployés par la même personne, ni en même temps. Le concept de grille de ressources de calcul est souvent comparé au concept de réseau de distribution électrique, dont on profite en se branchant via un simple connecteur, sans se soucier d'où provient cette électricité. L'analogie s'arrête souvent là, car, concrètement, il peut être nécessaire de respecter un critère de co-localisation des ressources de calcul, pour y placer des activités parallèles fortement couplées, communiquant de manière intensive. Nous pensons que c'est le programmeur qui est le plus à même d'exprimer ces contraintes. C'est pour toutes ces raisons que nous pensons qu'il est souhaitable que ce soit l'utilisateur final qui puisse orchestrer le déploiement de son application, donc, contrôler l'acquisition des ressources de calcul et les approvisionner avec le code et les données nécessaires, en fonction de contraintes spatiales et temporelles.

> Pour autant, nous pensons qu'il est important de ne pas (trop) polluer le code fonctionnel avec des opérations explicites de gestion de l'environnement d'exécution, ou de dépendances à son égard.

Comme présenté dans [C8], résumé en [Section 2.2.2, page 139, section 4], et récemment étendu avec des solutions de transferts de fichiers [W19,C20], ProActive résoud ce problème, d'une manière originale, et nous le pensons, élégante : le programme ne manipule en son sein que la notion de nœud virtuel, notion logique correspondant à un lieu d'exécution (JVM ProActive) sur lequel placer des activités (objets actifs). Les primitives proposées permettent au programmeur de déclencher le démarrage ou la découverte de ces JVMs par la simple demande d'activation de ces nœuds virtuels. Toute l'information nécessaire pour expliciter où et comment démarrer ou localiser ces exécutifs se trouve confinée à l'extérieur du code, dans des *fichiers de déploiement* au format XML. Le lien nécessaire entre ces informations et leur usage implicite se résume aux noms de nœuds virtuels (à la manière des rangs de processus MPI). On peut néanmoins se poser la question : est-ce suffisant ? En effet, certains travaux autour des objets parallèles et répartis, par exemple l'environnement ParoC++ [63] permettent d'associer au niveau même du code de l'objet, la description des besoins en ressources pour l'exécution. Nous pensons que notre solution est suffisante. En effet, il semble que rien n'empêche d'indiquer ces besoins en les associant à la description des nœuds virtuels (dans le fichier décrivant le déploiement de l'application et de son infrastructure d'exécution). Comme nous l'évoquerons brièvement dans la section 3.3 dédiée à l'extension de cette approche objets vers des composants, c'est via des meta-données associées au packaging du composant et non dans le code source lui même que l'expression de tels besoins semble la plus naturelle.
Bien sûr, toute notre mécanique capable d'interpréter ces informations de déploiement et d'agir en conséquence est totalement ouverte, extensible : il suffit de rajouter un nouveau `Process` pour déployer dans un nouvel environnement, y compris un environnement géré via un intergiciel de grille[6]. Contrairement à l'utilisation d'APIs éventuellement complexes d'utilisation si elles veulent pouvoir refléter toute l'étendue des options de déploiement (comme GAT [Section 2.2.4, page 181, section 3.3]), le mécanisme proposé est beaucoup plus simple d'utilisation, tout en étant puissant.

Notre approche est complémentée par un outil de surveillance, de pilotage, voire de déboggage, du déploiement et de l'exécution d'applications ProActive. Cet outil se nomme IC2D (Interactive Control and Debug for

[6]comme réalisé avec succès récemment, à l'occasion du 2nd Grid PlugTests and Contest d'octobre 2005, en interfaçant deux intergiciels de grille incontournables de plus, Unicore et EGEE gLite

Distribution), ses principes sont présentés dans [C6,C8] et résumés en [Section 2.2.2, page 139, section 4.3].

## 3.3 Extension vers une approche par composants logiciels

Il est reconnu que les composants logiciels exhibent des qualités intéressantes que ne présentent pas les objets. C'est tout particulièrement les suivantes qui nous paraissent pertinentes pour la programmation parallèle et répartie :

- Connexion dynamique des interfaces des composants (interfaces serveur ou client).
- Séparation naturelle des caractéristiques fonctionnelles et non fonctionnelles.
- Packaging de composants, avec livraison *sur l'étagère*, incluant l'utilisation de méta informations pour décrire les fonctionnalités du composant (et donc imaginer de pouvoir les sélectionner dans des catalogues de composants [W17]).

Nous présentons ci-dessous les différentes directions et résultats auxquels nous avons apportés notre contribution, et qui ont réellement pris forme grâce au travail de Matthieu Morel, d'abord en tant qu'ingénieur, puis en tant que doctorant.

### 3.3.1 Composants distribués

En tout premier lieu, le domaine du parallélisme nécessite que les composants soient eux-mêmes des codes parallèles et répartis. Dans notre contexte, il est apparu évident qu'un système d'objets actifs ProActive puisse constituer le code d'un composant, même si on privilégie un de ces objets comme étant le point d'entrée du composant. Ce système d'objets actifs peut lui-même être structuré sous la forme d'un programme OO-SPMD. Des travaux en cours réfléchissent à des solutions génériques à base d'objets actifs, pour emballer des codes patrimoniaux parallèles (par exemple écrits en C plus MPI). Même dans ce cas, le composant reste considéré comme un système d'objets actifs. De ce fait, il hérite naturellement de toutes les qualités intrinsèques d'une application ProActive, et principalement : traitement asynchrone des demandes d'exécution de services sur le composant, avec gestion automatique des réponses ; déploiement du composant sur un environnement

réparti, aussi complexe et hétérogène que peut l'être une grille ; migration possible du composant.

### 3.3.2 Composants répartis hiérarchiques

Un composant est naturellement accessible à distance. La création d'une application par assemblage de composants déployés par exemple sur une grille, est de ce fait naturelle. On peut trouver de nombreux frameworks à composants qui permettent cela. En particulier, les frameworks qui implantent le modèle à composants CCA [1], et qui ont apporté des solutions pour que les composants soient des codes parallèles. Cela comporte entre autre l'introduction de techniques adaptées pour la redistribution efficace de données entre ces différents codes parallèles [85], connues sous le terme générique de `MxN problem`. Néanmoins, l'assemblage de composants est plat, et donc pas particulièrement structuré.

> Nous avons contribué à promouvoir l'idée que l'assemblage de composants présente des avantages à être hiérarchique. Ceci permet de mieux appréhender les préoccupations non fonctionnelles, qui sont exacerbées dans le contexte de la programmation et l'exécution parallèle et répartie, en particulier sur grille de calcul.

Il nous semble en effet intéressant de tenter le parallèle entre la façon dynamique, spontanée, voire autonomique, qui sous-tend l'organisation et l'émergence de grilles de calcul, et les logiciels (composants ou plus généralement services) qui constituent la partie émergée et utile pour profiter des infrastructures sous-jacentes. Ces logiciels doivent pouvoir être le résultat de compositions à la fois `dans l'espace` (chaque composant est déployé sur des ressources de calcul de la grille, éventuellement éloignées géographiquement) et `dans le temps` (un service offert par un composant n'est pas forcément utile tout au long du cycle de vie d'un autre composant agissant comme client [35]). De plus, ces compositions doivent pouvoir évoluer dynamiquement, en fonction des instances disponibles, du fait par exemple de la volatilité des ressources (d'où l'importance d'une approche par composant plutôt que par objets uniquement).

Un modèle de composants hiérarchiques où dès le départ, toute composition de composants a elle même un statut de composant, apporte des qualités d'encapsulation et donc de délégation des actions et des prises de

décision. Ces qualités nous semblent utiles pour maitriser à la création, au déploiement, et à l'exécution :

- l'ordre de grandeur (plusieurs milliers de composants élémentaires peuvent contribuer à l'émergence d'un service applicatif) ;
- la complexité intrinsèque liée au contexte de grille. Ce contexte est particulièrement exigeant du fait de l'hétérogénéité des supports matériels et logiciels et de leur niveau de performance, du faible degré de fiabilité et de sécurité, de la volatilité des ressources, etc.
- cet ordre peut s'avérer si grand qu'une approche autonome dans les prises de décisions, qu'elles soient de niveau fonctionnel ou non fonctionnel se justifie (comme déjà exploré dans [6], [81], [50]).

Par exemple, supposons qu'une opération de maintenance soit planifiée sur un cluster de machines. Pour ne pas interrompre l'application, il est nécessaire de pouvoir lui imposer de migrer vers un autre cluster. L'opérateur d'un système hiérarchique de composant peut par exemple, de manière toute naturelle, faire se propager l'information de maintenance depuis la racine de la hiérarchie de composants constituant l'application. Un contrôleur de migration par composant hiérarchique, peut alors recevoir et décider de la nécessité ou non de propager l'information aux composants qu'il contient, et ainsi de suite. Au bout du compte, seules les instances utilisant effectivement les machines de ce cluster devront réellement réagir à l'information reçue et ce si possible, de façon coordonnée. A nouveau, cette prise de décision coordonnée peut utilement être réalisée en se reposant sur le niveau hiérarchique immédiatement englobant. En effet, c'est probablement parce que les instances communiquaient intensément qu'elles avaient été placées sur le même cluster. Le choix de leur nouvel emplacement doit donc respecter cette contrainte. Par conséquent, c'est naturel que ce soit le composant hiérarchique englobant qui ait la charge de piloter l'acquisition d'un nombre suffisant de ressources co-localisées sur un même cluster, puis de déclencher l'ordre de déplacement de toutes les instances.

**Composants hiérarchiques Fractal pour la grille** Depuis quelques années émerge une importante communauté de chercheurs autour de la définition et des implantations d'un modèle de composants hiérarchiques, Fractal [15]. Les implantations connues ne permettent pas d'avoir des composants qui soient à la fois parallèles et répartis, et où les invocations de services entre composants se prêtent bien à un environnement à large échelle telles

les grilles. Une implantation du modèle Fractal au dessus de ProActive a donc été entreprise [R8,R12,C12,C15]. Tout comme n'importe quel composant primitif, un composant hiérarchique est implanté par le biais d'un objet actif ProActive (voir Section 2.3.1, page 201+8, section 4]), et la partie MOP de l'objet est étendue pour implanter la spécification Fractal.

### 3.3.3 Problèmes spécifiques liés à la grille

Nous avons pu identifier et commencer à résoudre un certain nombre de points liés à cette proposition.

**Membranes** Dans le modèle Fractal, tout composant qu'il soit primitif ou hiérarchique comporte une partie de contrôle (membrane). Toute invocation de service émanant d'un composite (entrante ou sortante) nécessite donc de traverser sa membrane, ce qui se traduit par un appel vers un objet actif, vers lequel le temps de communication n'est jamais négligeable dans le contexte de grilles. Il serait souhaitable de pouvoir court-circuiter les membranes des composites (établir des raccourcis), tant que des demandes de reconfiguration d'un composite ne sont pas en cours de traitement. En effet, les liaisons établies dans la membrane du composite peuvent être remises en cause lors des reconfigurations. Il faut donc être capable dynamiquement d'annuler les raccourcis pour en rétablir plus tard. Il faut prendre garde à conserver l'ordre FIFO des communications du système ProActive sous-jacent du fait de ces différentes annulations et rétablissements de chemins suivis par les messages ProActive.

**Déploiement** Il est possible de spécifier dans le fichier de méta informations associées au composant un nom de nœud virtuel au sens habituel ProActive. On envisage également d'ajouter d'autres informations quant aux besoins en ressources pour l'exécution du composant. Lors de l'assemblage d'un système à composants, il faut donc également permettre la composition de ces informations [section 3.2 Section 2.3.1, page 201+8].

**Composants hiérarchiques parallèles** L'assembleur du système de composants devant s'exécuter sur la grille peut vouloir profiter du parallélisme matériel sous-jacent. Deux possibilités existent : un composant primitif est déjà conçu comme un code parallèle, ou un même composant primitif peut

donner lieu au déploiement de plusieurs instances. Dans ce second cas, en accord avec l'approche Fractal, il est naturel de réunir toutes ces instances au sein d'un unique composite, que l'on qualifie de composant hiérarchique parallèle à déployer sur un ensemble de ressources [Section 2.3.1, page 201+6]. Mais dans ce cas, au lieu d'invoquer le service sur un seul réplica, on veut pouvoir l'invoquer en parallèle (c'est-à-dire efficacement) sur tous les réplicas. Les données nécessaires pour ce service peuvent elles-mêmes être décomposables, de manière à ce que chaque service reçoive et traite en parallèle une portion de ces données. Un réplica peut avoir besoin de faire appel à un service, et du coup, l'ensemble parallèle de réplicas aussi. Chaque réplica peut indépendamment des autres effectuer cet appel. Il peut s'avérer plus efficace de ne faire qu'un unique appel pour l'ensemble, et une fois le résultat renvoyé, de le transmettre à chaque réplica après découpage éventuel. Il faut donc étendre le modèle standard de service offert et service requis pour satisfaire ces nouvelles spécifications et les mettre en œuvre au sein du framework. Il a donc été proposé d'introduire les notions d'interfaces collectives serveur ou client, de type *multicast* (un appel se transforme en un ensemble d'appels), ou *gathercast* (pour synchroniser un ensemble d'appels afin de n'en réaliser qu'un seul) [R13,C18]. Lier deux composants parallèles composites (voir figure 3.2) peut alors nécessiter de résoudre un problème similaire au `MxN problem` qui se pose lorsque l'on relie deux composants primitifs qui sont eux-mêmes des codes parallèles. Conformément à l'approche Fractal, les composants, qu'ils soient primitifs ou composites doivent avoir la même spécification. Une spécification unique applicable aux deux cas s'avère nécessaire [R16].

> Le bilan est que l'on essaye de créer du parallélisme non pas par programmation mais par composition.

On est bien conscients des limites actuelles : les replicas ne se connaissant pas, ils ne peuvent pas interagir entre eux. Nous avons pu le constater en étudiant dans [R18] la transformation de l'application Jem3D fondée sur des groupes d'objets actifs typés, en une version à base de composants. Une extension plus profonde de l'approche est donc nécessaire. Par exemple, un composant hiérarchique parallèle organiserait les réplicas qu'il contient selon un groupe OO-SPMD. Les relations nécessaires entre les réplicas pourraient reposer sur des structurations topologiques des objets actifs du groupe (selon [Section 2.2.3, page 171, sections IV.B, IV.C]) et l'ADL Fractal étendu pour désigner les relations de voisinage.

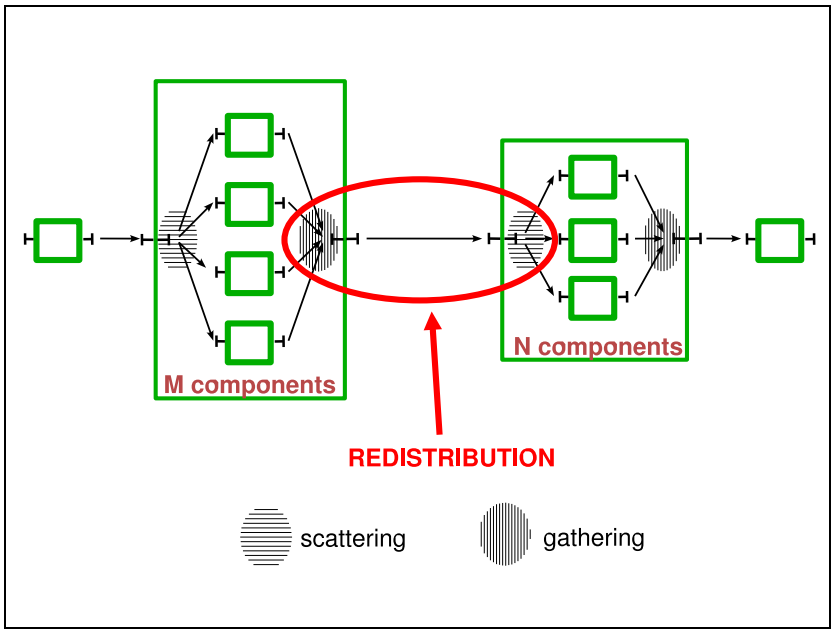



FIG. 3.2 – Couplage de deux composants hiérarchiques et parallèles

**Interopérabilité des composants avec des services** Dans un monde ouvert comme l'est une grille, les applications qu'elles soient simples ou complexes, ont vocation à être publiées pour pouvoir servir à la communauté toute entière. Notre vision est la suivante : les composants répartis hiérarchiques sont utiles pour la définition et la mise en œuvre d'un service, lorsque sa logique et sa structure applicative et non-fonctionnelle sont complexes[7]. Il a ensuite vocation à être exposé et publié selon la technologie à services désirée : en tant que service Web [W18,B3], ou selon toute autre technologie à services, par exemple en tant que service OSGi (voir 4.2) réparti, etc. Symétriquement, la mise en œuvre d'un service par l'approche composants devrait pouvoir tirer profit des services existants et publiés dans l'environnement dans lequel le service est plongé à l'exécution (voir par exemple [7] où les services requis de composants GRID.it peuvent être liés à des services Web).

[7]La structuration globale du système à composants peut d'ailleurs s'organiser selon des squelettes, comme étudié dans [28]

## 3.4 Conclusion

Dans ce chapitre, nous avons dressé un panorama des différentes contributions apportées aux recherches effectuées avec comme point de départ le concept d'objet actif accessible à distance.

Au delà des extensions spécifiques apportées au concept d'objet actif (mobilité, communication de groupe, etc.) c'est l'intégration de toutes ces caractéristiques additionnelles qui constitue aussi une difficulté[8]. Mais, en même temps, c'est ce qui nous semble constituer la force de l'approche et des solutions proposées. Rares sont les plateformes de programmation et d'exécution parallèle et répartie qui peuvent prétendre offrir aux applications un tel continuum dans les services techniques : des aspects liés à l'assemblage et à la programmation, jusqu'au déploiement et le suivi de l'exécution dans les cadres les plus exigeants que sont les grilles. Evidemment, d'autres environnements ont les mêmes types de prétentions, mais ils réussissent peut-être moins bien à fédérer tous ces aspects, et à être d'application aussi générale.

En particulier, les objets actifs et les composants permettent d'exprimer des schémas de communication quelconques. Les plateformes intégrées pour la programmation des grilles privilégient souvent des modèles plus simples style "sac de tâches" qui n'interagissent pas pendant leur résolution (OurGrid [10], Alchemi développé dans le contexte du projet Gridbus [52], ...).

Ou bien, les solutions proposées se cantonnent au rôle de bibliothèque entre processus répartis, véhiculant des messages, proposant un service de nommage (ex. PVM, MPI, Voyager [40]) ; ces bibliothèques sont alors portées sur telle ou telle technologie ou infrastructure répartie (MPICH-G2 [44] est le portage de MPI sur Globus, [60] présente un plug-in PVM sur Harness, etc.).

A l'autre bout du spectre, on trouve des environnements ou middlewares permettant "juste" de fédérer des ressources au sein d'une grille de calcul pour ensuite y soumettre des jobs : Unicore [78], tous les nombreux middlewares bâtis au dessus de Globus [32], Legion [49] qui se distingue par sa modélisation tout objet des ressources et services nécessaires pour bâtir la fédération, ....

Le projet H2O [46] semble être celui qui se rapproche le plus de notre démarche intégrée. H2O propose une approche à objets distribués pour pro-

[8] par exemple, l'implantation des barrières de synchronisation OO-SPMD agit sur l'ordre des requêtes dans la queue ; il faut qu'elle soit compatible avec le protocole de tolérance aux pannes par sauvegarde de l'état et reprise par recouvrement, qui se fonde sur une gestion fine de l'état de la queue des requêtes [C19]

grammer des applications, ensuite déployées sur un runtime portable et très modulaire via une approche plug-in pour les aspects non fonctionnels. Le modèle de programmation sous-jacent n'est cependant pas si riche (pas de future automatiques, pas de groupes d'objets typés permettant de la communication 1-N, ...). Il existe aussi au dessus d'H2O une implémentation d'un modèle à composants, le modèle CCA [53]. Une intéressante collaboration, rendue possible grâce au réseau d'excellence CoreGRID, s'est imposée : l'objectif est de pouvoir inclure des composants CCA (au dessus d'H2O), au sein de composants composites Fractal-ProActive (qui du fait de leur approche hiérarchique sont donc plus généraux)[D3]. La motivation globale est d'aboutir à un modèle de programmation par composition suffisamment solide et riche, pour applications visant à être déployées et exécutées sur grilles. Une approche par composants logiciels est de ce fait naturelle. Ce genre de collaborations présente l'opportunité de confronter, enrichir mutuellement, voire rendre interopérables les plateformes de programmation sur grille existantes. L'effort d'interopérabilité semble pertinent dans un monde de ressources et services globalisés tel que promu par le concept de grille : l'idée est d'étendre ce concept aux applications.

# Chapitre 4

# Application à l'administration

La synthèse présentée dans ce chapitre permet d'illustrer dans quelle mesure les mécanismes d'extension de langages objets pour le parallélisme et la répartition disponibles en particulier dans ProActive, sont utilisables dans un contexte plus large que le "simple" calcul scientifique haute performance. Au contraire, on va voir la pertinence de leur utilisation dans un domaine bien différent : l'administration.

## 4.1 Administration système et réseau

### 4.1.1 Contexte général et Problématique

Depuis déjà de quelques années, l'exploitation d'agents mobiles pour réaliser des opérations d'administration de systèmes et de réseaux, lorsque ceux-ci sont de grande taille, est sérieusement étudiée. En effet, l'approche par agent mobile a le potentiel de réaliser ces opérations par délégation en lieu et place de l'administrateur en allant au delà d'un modèle plus classique "client-serveur". Il y a bien d'autres bonnes raisons d'utiliser le paradigme à agent mobile, et plus basiquement, le paradigme acteur ainsi que celui à *objet mobile* pour construire des applications réparties. Parmi les 7 bonnes raisons couramment évoquées [48], les suivantes sont celles que nous exploitons :

- réduction du trafic réseau,
- exécution asynchrone et autonome.

Après un certain succès, puis un certain abandon, le concept d'agent mobile semble redevenir populaire. Par exemple, du fait de l'émergence de *desktop*

*grids*, on peut envisager que l'exécution des calculs se fasse via une plateforme logicielle à agents mobiles, les agents acheminant les calculs sur les ressources disponibles [34]. Plus généralement, le concept d'agent mobile peut supporter le principe de *Distributed Sequential Computing*, ce dernier consistant à avoir des fils d'exécution séquentiels, mais répartis, et capables de migrer afin de poursuivre leur exécution, par exemple en se rapprochant au plus près des données à traiter [65].

Nous exploitons ces qualités escomptées des agents mobiles, pour opérer des systèmes et des réseaux, nécessitant de récolter et disséminer de l'information sur ces systèmes, éventuellement en paralléle.

Le protocole d'administration de base utilisé pour l'administration de systèmes et de réseaux est le SNMP (Simple Network Management Protocol) permettant de récolter d'une manière standard les informations pertinentes pour les tâches d'administration. Le cas typique d'utilisation d'un agent mobile d'administration est celui où l'agent visite des systèmes et des équipements réseau et sur chacun, réalise des collectes d'information via SNMP, qu'il aggrège et analyse de manière autonome, pour ensuite les tenir à disposition de l'administrateur. L'administrateur peut interroger l'agent à distance, ou localement, une fois ce dernier parvenu à la fin de son itinéraire et revenu par exemple à la station de base d'où l'administrateur l'a lancé. L'intérêt numéro un réside dans le fait d'une diminution potentielle de la bande passante réseau nécessaire pour rendre disponibles ces informations à l'adminstrateur : étant collectées et fusionnées sur place, la charge de communication induite se concentre uniquement sur le résultat de ces fusions (qu'on espère être de taille plus réduite que si les informations brutes étaient remontées intégralement à la station d'administration) ; un autre avantage potentiel est l'utilisation d'un ensemble d'agents qui permettent de tirer partie de collectes réalisées en parallèle. Ces agents peuvent même se concerter par le biais d'interactions, afin de ne présenter à l'administrateur qu'une synthèse globale de leurs opérations, diminuant potentiellement le goulot d'étranglement que représente une unique station d'administration.

Un certain nombre de plateformes issues de la recherche académique ont été proposées. Elles définissent un cadre de programmation pour des agents mobiles dédiés à des opérations d'administration système et réseau (voir par exemple MAP [67], MobileSpaces [69]). Dans ce contexte, il nous a semblé pertinent d'explorer dans quelle mesure notre approche et outil de programmation répartie et mobile (ProActive) pouvaient être applicables pour la supervision de systèmes et de réseau, et quels pourraient en être les bénéfices

par rapport aux autres.

### 4.1.2 Plateformes d'administration par agent mobile

#### Difficultés de programmation

Parmi les points délicats relatifs à la programmation d'agents mobiles pour l'administration, nous avons tout particulièrement considéré les suivants :

- le besoin pour un agent d'effectuer parfois des opérations SNMP, parfois des opérations exprimables uniquement en langage Java, et ce, en fonction des caractéristiques de l'élément cible
- le besoin pour un agent de connaitre et dérouler son itinéraire de visite, et que ce dernier soit à jour, sachant que plus le réseau est de grande taille, plus la liste des éléments le constituant est sujette à des évolutions.

Notre proposition pour prendre en compte ces deux points, et ce de manière intégrée, se trouve décrite plus amplement en [C9] (voir [Section 2.4.1, page 221]) et [C11] ([Section 2.4.2, page 235]), et synthétisée ci-dessous. Ce travail a constitué la thèse de doctorat d'Emmanuel Reuter.

#### Maintien et mise à disposition de la topologie du domaine à administrer

Le premier volet repose sur la découverte dynamique de la topologie du réseau dont la plateforme d'administration par agents mobiles est en charge [Section 2.4.1, page 221+4, section 2.2] et [Section 2.4.2, page 235+2, section III.A]. Pour cela, nous avons programmé un `DiscoveryAgent` (lui-même un agent mobile) dont la tâche est de récolter, et combiner des informations sur les différents éléments présents, et la manière dont ils sont interconnectés. Ces interconnections sont celles existantes au niveau 2 des couches OSI, donc, reflétant précisément l'organisation réseau sous-jacente. Par ailleurs, les nœuds d'accueil potentiel d'agents mobiles (runtime ProActive) pouvant être présents sur les hôtes (PCs) rencontrés sont enregistrés lors de ce processus de découverte.

Ces informations sont collectées au niveau de chaque LAN constituant l'ensemble du domaine d'administration. Ensuite, ces informations sont stockées et rendues disponibles dans des serveurs d'itinéraires (en général, un serveur

par LAN). Ces serveurs d'itinéraires sont de plus enregistrés en tant que service Jini, pour être ultérieurement retrouvés par des agents mobiles devant parcourir le domaine, dans son intégralité ou partiellement selon un certain critère (par exemple, selon le type de l'équipement, imprimante, routeurs, etc) (voir [Section 2.4.1, page 221+6, section 3.2]).

### Itinéraire générique d'administration

Le second volet de notre proposition est décrit ci-dessous et se penche sur la question suivante.

> Pour exploiter pleinement la puissance des agents mobiles dans le contexte de l'administration, il nous faut rendre flexible la manière d'exprimer et d'exécuter la tâche d'administration confiée à un agent mobile.

Pour cela, nous étendons le mécanisme de gestion d'itinéraire pour objets mobiles qui est disponible de base dans ProActive : ce mécanisme permet à un objet actif mobile de disposer d'un itinéraire et de pouvoir le suivre de manière automatique. Le principe de cette extension est de représenter par sous-classement les nouveaux types de destination nécessaires dans notre contexte de l'administration (voir [Section 2.4.1, page 221+6, figure 2 et section 3.1]). En plus du classique `NodeDestination` qui permet d'accueillir un objet actif ProActive sur un runtime ProActive, nous avons introduit en particulier deux nouveaux types de destination : `SNMPdestination` et `ItineraryServerDestination` [Section 2.4.2, page 235+3, section IV.B]). Les principes sont les suivants :

- une `SNMPDestination` permet de décrire quelles opérations effectuer selon le protocole SNMP (en interrogeant l'agent SNMP) concernant l'élément courant de l'itinéraire "visité" par l'agent ; cette interrogation pourra se faire soit à distance ou localement. Pour être fait localement, l'agent aura préalablement migré sur l'élément, en accord avec son itinéraire. Bien sûr, cet élément doit hégerger un nœud d'accueil ProActive.
- une `ItineraryServerDestination` permet d'insérer dans l'itinéraire suivi par l'agent mobile, une opération d'interrogation des prochains éléments à rajouter à l'itinéraire. C'est de cette manière que l'on est capable de construire des itinéraires non statiques puisque c'est seulement au moment de l'interrogation de ce serveur, que l'agent récoltera

une liste a priori à jour, d'éléments à visiter, qui seront alors rajoutés explicitement à son itinéraire.

Ainsi, cette manière originale de fabriquer des itinéraires d'administration permet de simplifier la tâche de l'administrateur du domaine. Ce dernier spécifie par programmation le type d'itinéraire désiré : il le fait en ayant juste à indiquer le type de gestionnaire d'itinéraire `ItineraryManager` qu'il veut voir associé à l'agent, et lui passe en paramètre le serveur d'itinéraire à utiliser, au moins initialement (voir [Section 2.4.1, page 221+8, section 4, et en particulier les figures 4 et 6] et [Section 2.4.2, page 235+4], figure 2). L'agent mobile est par contre totalement autonome en ce qui concerne le suivi automatique de son itinéraire et des opérations associées à exécuter sur chaque élément "visité", selon le type de cet élément.

Disposant d'un tel socle, on pourrait rendre encore plus flexible la prise de décision suivante : est-il plus performant de faire migrer l'agent, ou vaut-il mieux lui faire réaliser des opérations SNMP distantes [79], soit depuis la station d'administration, soit depuis l'endroit où il se trouve actuellement. Cette décision se traduirait alors par la sélection du type de Destination approprié pour chacun des équipements à administrer.

> Nous avons ainsi généralisé la notion standard d'itinéraire connue dans le contexte des plateformes à agents mobiles. Un itinéraire ne contient pas uniquement des indications de lieux où l'agent doit effectivement migrer pour y effectuer localement une action Java. Au contraire, ces lieux correspondent à n'importe quelle étape dans le déroulement du travail de l'agent, que cette étape implique une migration effective ou non.

Nous n'avons exploré que des enchainements séquentiels dans le déroulement de ce travail, mais, évidemment, toute extension est possible. Comme expliqué dans le manuscrit de la thèse de doctorat d'Emmanuel Reuter, on peut tout à fait étendre cette notion pour des itinéraires de visite parallèle : il suffit d'insérer dans l'itinéraire un nouveau type de destination sur laquelle l'agent pourra se clôner, chaque clône se chargeant par exemple de la visite d'un sous-ensemble du domaine particulier ; et on peut ensuite rajouter à chaque fin de sous-itinéraire ainsi que dans l'itinéraire de l'agent principal, un type de destination servant à réaliser une synchronisation des agents clônés (à la manière d'un *fork-join*).

### 4.1.3 Conclusion

Cette "incursion" dans le domaine de recherche lié à l'administration système et réseau s'est plus particulièrement concentrée sur la supervision d'infrastructures en réseau (et donc peut naturellement s'appliquer aux environnements de type grille [80]) : celà nécessite des moyens à la fois portables et efficaces pour récolter l'information, et l'exploitation du concept d'agent mobile s'imposait donc. Notre effort s'est porté sur la définition de mécanismes pouvant simplifier le travail du programmeur, en favorisant flexibilité et réutilisation de code existant dans la définition des tâches d'administration répartie confiées aux agents.

De manière cohérente avec l'ensemble de notre démarche, ces mécanismes se veulent simples d'usage, tout en étant suffisamment ouverts pour s'adapter à de nombreuses situations, et sans perdre de vue la possibilité d'exploiter du parallélisme si cela s'avère pertinent. Comme nous allons l'expliquer dans la prochaine section, c'est encore cette démarche que nous avons suivie, en nous penchant à nouveau sur une problématique liée à l'administration : celle de parcs d'infrastructures logicielles, avec comme point de mire *la maitrise de la grande échelle* dans les opérations de déploiement de logiciels, leur supervision, ainsi que celle de leurs infrastructures d'accueil.

## 4.2 Administration de plateformes à services

### 4.2.1 Contexte général et Problématique

L'informatique devient pervasive, connectée, donc communicante et répartie : elle a envahi nos maisons, nos véhicules, notre environnement, la plupart des lieux collectifs, et ce sous la forme de services déployables, accessibles à distance. Cette explosion à la fois de lieu et de masse requiert des moyens appropriés d'administration : le passage à l'échelle dans les opérations d'approvisionnement et gestion des services, ainsi que de surveillance des plateformes matérielles et logicielles qui les hébergent est un point clé. Notre objectif est d'explorer l'utilisation de techniques issues du parallélisme dans ce cadre. Plus précisément, le travail que nous sommes en train de réaliser est financé par le Réseau National de Recherche en Télécommunications, par le biais du contrat nommé PISE : Passerelle Internet Sécurisée et flexiblE.

Comme le résume le site web du RNRT, l'objectif de PISE est de *concevoir, développer et valider une infrastructure logicielle pour passerelles In-*

*ternet sécurisées capable d'accueillir dynamiquement des services techniques et métier, ainsi qu'un outil d'administration distante pouvant gérer un parc comportant des milliers de passerelles.*

Concrètement, cette vision repose sur l'utilisation de plates-formes logicielles ouvertes, reliées à l'Internet, jouant si besoin le rôle de passerelle avec un réseau local. Ces passerelles sont ouvertes, car capables d'héberger des applications sous formes de services provenant d'entités légales éventuellement différentes, et permettant si nécessaire d'obtenir des applications à très haute valeur ajoutée par composition et partage de tels services. Des initiatives pilotées par le monde industriel ont vu le jour afin de concrétiser ce type de vision. Nous nous intéressons dans la suite à l'initiative OSGi : Open Services Gateway Initiative [36], et reprenons en partie la description qui en est faite dans [33].

OSGi[1] est une proposition qui définit les API nécessaires pour pouvoir exécuter et gérer des services sur une passerelle. L'API OSGi repose sur la machine virtuelle et le langage Java. Elle peut brièvement se résumer selon les trois points suivants :

- Le conteneur de services : c'est un démon Java qui garantit l'exécution des différentes briques logicielles hébergées. Il autorise et permet l'association entre des (services locaux) clients demandant l'accès à des services et des briques implantant ces services. En résumé son rôle est d'enregistrer et de gérer localement l'activité de la plateforme.
- Les services standards : la spécification définit un certain nombre de services standards (http, logging, console, ...) que la plateforme peut proposer. Ces services sont directement exploitables par d'autres services.
- Un modèle de livraison de code : le modèle repose sur le concept de *bundle* comme unité de transport et de déploiement de classes, et de ressources, qui peuvent être ensuite partagées sur la passerelle. C'est une archive .jar qui peut être téléchargée puis installée par la passerelle, et ce de manière continue (sans stopper la passerelle). La description du contenu du bundle se fait par un fichier de description au format Manifest Java. En plus de ressources usuelles (packages .jar, bibliothèques natives, ...), un bundle OSGi peut contenir du code qui une fois démarré peut rendre des services aux autres services s'exécutant sur la passerelle. Ce partage de packages et de services permet d'envisa-

[1] www.osgi.org

ger de réduire l'empreinte mémoire totale d'un ensemble d'applications, et rend ainsi possible leur hébergement sur des passerelles OSGi embarquées sur des équipements à ressources physiques limitées, tels des équipement mobiles, des ordinateurs embarqués dans des véhicules.

Le conteneur gère le cycle de vie des bundles et des services : il installe les bundles, résoud les dépendances entre bundles en fonction de leurs importations et exportations de packages, permet au bundle de s'activer ou se désactiver en fonction de ces dépendances. Cette dynamicité se prête bien à l'application d'une vision orientée services : l'activation de bundles engendre des objets dont les interfaces peuvent être considérées comme des services dès lors qu'elles sont enregistrées dans un registry local à la passerelle. Des notifications sont automatiquement générées par le conteneur pour prévenir de l'apparition ou disparition de ces services : il s'agit d'une approche par services dynamiques. Par conséquent, ce modèle implique pour le programmeur la mise en place explicite d'écouteurs pour les services dont on veut surveiller le cycle de vie. Afin d'abstraire ce style de programmation quelque peu délicat et enclin à mélanger la logique applicative et la prise en compte des spécificités liées à OSGi, une vision 'composants' a été récemment proposée [21] : elle a pris la forme d'un nouveau service OSGi, *Service Binder*, dont le rôle est de prendre entièrement en charge le cycle de vie et les liaisons entre les services de la passerelle à partir uniquement d'une description simple et déclarative des services offerts et requis par un composant (en l'occurence, la notion de composant devenant associée au concept de bundle). Ce concept est disponible dans la dernière spécification OSGi (version 4), sous le terme officiel de SCR (Service Component Runtime)[2].

La spécification OSGi ne propose par contre aucune solution 'standard' concernant son administration et sa supervision à distance. La spécification s'arrête essentiellement aux API pour la gestion du cycle de déploiement et l'exécution des services. Cette limitation est un large frein à l'utilisation d'une telle plateforme comme conteneur de services, gérée par un, voire plusieurs opérateurs simultanément. Ces opérateurs sont par ailleurs à distance, et sont potentiellement en charge de centaines voire milliers de telles plateformes.

[2]Une extension naturelle de OSGi, et de Service Binder, concerne l'aspect réparti des services : pouvoir utiliser un service OSGi distant (c'est un des points abordés par nos partenaires dans PISE, et que nous mêmes commençons à explorer dans le cadre d'un autre contrat, le projet ITEA S4ALL *Services for All*, mais que nous laissons volontairement de côté ici).

Dans ce cadre, Virginie Legrand, ingénieur recruté sur contrat PISE, et moi-même, nous nous sommes attaquées à la problématique du passage à l'échelle dans les opérations de supervision.

Comme point de départ, nous nous sommes inspirées de travaux réalisés notamment dans le cadre du projet ITEA OSMOSE[3] ou [33], proposant d'appliquer l'outil JMX (Java Management eXtensions) défini par Sun concernant la gestion d'applications ou plus largement, d'infrastructures logicielles, au cas des passerelles et applications OSGi. Contrairement à SNMP plus approprié à la gestion de systèmes et de réseaux, le choix de JMX est plus pertinent puisque intimement lié à Java, et donc à la supervision d'applicatifs.

JMX, et son extension pour l'accès aux fonctions de supervision à distance, la JSR 160, définit une architecture ainsi que les APIs correspondantes [30], organisée autour :

- d'une part, d'un ensemble de petits composants appelés Mbeans, associés à chacune des ressources ou objets à superviser.
- d'autre part un agent, en charge de gérer le cycle de vie et l'accès aux Mbeans, via un serveur appelé Mbean server, rendu accessible à distance par le biais de connecteurs interchangeables (connecteur HTTP, SNMP ou toute autre technologie d'accès distant via IP) et des adapteurs de protocoles (assurant la liaison entre des protocoles spécifiques comme SNMP ou http et les services locaux qu'offre l'agent, par exemple pour invoquer des méthodes sur les Mbeans).

En d'autres termes, le modèle d'administration JMX nécessite d'instrumenter les ressources logiques ou physiques à administrer, en y associant des Mbeans sur lesquels l'administrateur pourra ensuite agir par appel de méthode ; on peut aussi définir quelles actions réalisées sur l'applicatif supervisé devront être remontées à l'administrateur via la génération d'événements.

**Point de départ : Gestion JMX sur OSGi** Appliquer JMX à l'administration distante de plateformes OSGi passe par la livraison sous forme de bundles de l'agent, ainsi que des connecteurs voulus pour l'accès distant. Il faut aussi définir le ou les Mbeans qui seront gérés par l'agent. Nous avons choisi de n'en définir qu'un seul par plateforme OSGi. Il propose un ensemble minimal mais suffisant de méthodes pour récolter les informations de base et déclencher des opérations constituant un socle pour l'administration : en plus des classiques opérations d'interrogation permettant d'obtenir des pa-

[3] www.itea-osmose.org

ramètres ou propriétes de la machine, du système d'exploitation, de la passerelle OSGi, ce Mbean permet d'obtenir la liste des bundles installés, leur état, la liste des services publiés, et offre la possibilité d'installer de nouveaux bundles, d'agir sur leur cycle de vie, etc. Par ailleurs, l'API OSGi permet de notifier l'exécution d'opérations de base effectuées par la plateforme, telles que installation/desinstallation de bundle, démarrage, enregistrement, arrêt de service, etc. Nous avons donc décidé de faire remonter ces notifications à l'administrateur via des notifications JMX.

### 4.2.2 Besoins de l'Administration à grande échelle

L'utilisation d'une technologie telle JMX répond à un des critères primordiaux en terme d'administration : l'aspect distant de l'objet à administrer et de l'outil en charge de cette administration. Un autre aspect, de plus en plus présent du fait de la prolifération des équipements et services en réseau, relève de la multitude de tels objets à administrer. Un dernier aspect provient de la multiplication des sources d'approvisionnement de services, et donc de la nécessité de confinement des opérations d'administration en fonction des entités reconnues et autorisées.

> La prolifération d'équipements et services en réseau requiert de rechercher des solutions d'administration qui concilient répartition, asynchronisme (pour avoir du parallélisme dans l'exécution d'opérations), sécurité, et bien entendu, facilité d'utilisation.

Pour un administrateur, nous pensons que ce dernier critère se traduit par la capacité de son outil à refléter la structuration du domaine réel dont il a la charge (que ce soit un domaine physique, c'est-à-dire un parc de machines et d'équipements, ou un domaine virtuel constitué de services). Ce domaine peut s'avérer complexe, vaste, mais néanmoins, il est couramment organisé d'une manière hiérarchique, qui pour le cas d'un domaine virtuel peut de manière naturelle être calqué sur le domaine physique (notion de structure d'entreprise [59]).

Pour fixer les idées par le biais d'un exemple, imaginons un domaine physique, constitué de compteurs électriques déployés sur une zone géographique à l'échelle d'un pays. Ce domaine physique est lui-même plus facilement appréhendable par l'opérateur (installateur, ou réparateur des compteurs), grâce à une décomposition récursive en zones géographiques ou administratives plus petites. Sur un tel parc sont déployés des services de gestion de

consommation électrique, mais du fait de l'ouverture du marché à la concurrence, les offres de tels services peuvent provenir de fournisseurs différents. Considérons le point de vue d'un autre opérateur, celui en charge de superviser l'ensemble des services déployés, fournis par la compagnie A, ce qui constitue alors un domaine virtuel et non physique. Il peut lui aussi vouloir structurer ce domaine en suivant une structure hiérarchique fonction de la géographie sous-jacente. Mais il peut aussi avoir un intérêt à gérer ce domaine virtuel selon un autre regroupement : par exemple, selon une arborescence reflétant les différentes versions existantes pour un service donné déployé sur le parc.

Les caractéristiques des tâches demandées à un administrateur sur un tel domaine doivent également être reflétées dans l'outil. Ces tâches sont essentiellement de type surveillance ou action (installation, mise à jour, ou désinstallation), et orthogonalement, concernent l'ensemble ou un sous-ensemble du domaine, ou bien un élément en particulier.

Donc selon ces deux dimensions que sont le domaine et les activités d'administration, la tâche la plus complexe consistera à déployer une application ou service (lui-même obtenu par composition de plusieurs services plus élémentaires), sur chaque élément du parc ; en considérant de plus que ce parc est hétérogène du point de vue physique, et du point de vue de celui des services requis. En effet, certains éléments, mais pas tous, peuvent déjà disposer des services requis, et ce dans des versions peut-être différentes, mais nécessairement compatibles avec les services manquants qui y seront déployés. Réaliser un tel déploiement doit donc reposer sur un plan, que nous considèrons être préalablement calculé [4]. Ce plan décrit pour l'ensemble des éléments du parc concerné, l'ensemble de toutes les actions à réaliser individuellement sur chaque élément de l'ensemble. Pour calculer le plan, la plateforme d'administration doit maintenir un descriptif le plus précis et exact possible des domaines physiques et virtuels, car servant de paramètre d'entrée à ce calcul. En effet, le calcul se fonde sur l'estimation du différentiel qu'il y a entre l'état actuel et l'état voulu, et ce pour chaque élément. Une fois un plan calculé, l'opérateur doit pouvoir exécuter le plan via l'outil d'administration. Non seulement, l'exécution doit être rapide et en mesure de passer à l'échelle, mais aussi, présenter des qualités de type transactionnel [54] afin de pouvoir être validée ou annulée.

[4]Dans le contexte du projet PISE, c'est notre partenaire, l'équipe ADELE du LSR-IMAG qui est en charge de ce calcul [47]

Pour cela, nous faisons l'hypothèse d'une séparation claire entre deux rôles : celui qui commande le déploiement, celui qui exécute le déploiement. L'exécuteur doit donc faire remonter au commanditaire toutes les informations relatives au déroulement de l'exécution du plan : pour chaque élément, succès, ou échec, et en cas d'échec, jusqu'où le plan a pu être exécuté avec succès. Le commanditaire peut alors décider de demander l'annulation de l'exécution des opérations réalisées avec succès, afin de remettre l'ensemble du parc ciblé dans un état global antérieur (une sorte d'annulation de transaction, la transaction étant constituée de l'ensemble des opérations décrites dans le plan). Au contraire, il peut décider d'entériner l'exécution (une sorte de confirmation de l'exécution de la transaction), même si elle n'a pas entièrement réussi, quitte à initier ensuite certaines actions correctives (comme des installations, ou désinstallations) sur certains éléments du parc.

Dans la suite, nous considérons que l'exécution d'un plan de déploiement sur le parc se concrétise par l'exécution d'une **transaction**. Comme le parc ciblé contient plusieurs éléments, il s'agit d'une transaction **répartie**, et pour passer à l'échelle, nous ferons en sorte qu'elle s'exécute de manière **parallèle** (voir 4.2.7). Comme de plus on autorise (sous certaines conditions) que plusieurs plans de déploiements s'exécutent en même temps sous le contrôle d'un même administrateur, ces transactions sont **concurrente**s. La suite de ce chapitre décrit une solution pour de telles transactions.

### 4.2.3 Solution pour l'administration à grande échelle

Notre contribution à l'administration à grande échelle de passerelles OSGi, via JMX, se fonde sur ProActive et prend la forme d'un outil d'administration (dont les fonctionnalités et principes sous-jacents sont décrits plus précisément dans une première version de la documentation associée [R20]).

> Ce travail a aussi comme motivation d'illustrer l'intérêt de la technologie fondée sur le modèle à objet actif permettant des traitements et communications asynchrones et sécurisées, étendus à des groupes d'objets actifs de même type. Par ce biais, nous voulons aussi illustrer son adéquation à des domaines applicatifs moins habituels, comme l'est par exemple le calcul haute performance, mais où la mise en œuvre de techniques de parallélisme peut pourtant se justifier.

Notre solution se décline en plusieurs points que nous allons traiter individuellement. La description de ces points est assez longue, car, il n'existe pas encore de publication permettant de les présenter sous forme d'article inclus dans l'annexe (même si le projet PISE est décrit succintement dans [W23]). L'implantation des spécifications pour ces différents points en est au stade final, et permettra de complètement valider la démarche. Celle-ci s'articule selon la définition de :

- Groupes de connecteurs ProActive pour JMX
- Remontée de notifications JMX via ProActive
- Exécution transactionnelle d'un (ou plusieurs) plan de déploiement global
- Exécution parallèle d'un plan de déploiement global

L'articulation de ces points nous permet de proposer une solution d'administration à grande échelle pour passerelles OSGi.

### 4.2.4 Groupes de connecteurs ProActive pour JMX

La première brique consiste à réaliser un point d'entrée vers un environnement géré par JMX, en utilisant ProActive. Celui-ci diffère donc peu de l'accès via RMI, et cependant apporte quelques avantages surtout en terme de performances. En effet, ceci va permettre de déclencher de manière asynchrone l'exécution d'une opération d'administration sur un élément donné d'un parc. Pour celà, il suffit que l'opération prenne la forme d'une invocation de méthode sur un objet actif.

Nous avons donc défini un nouveau connecteur JMX pour l'accès distant, qui est en fait un objet actif ProActive, rendu accessible à distance par son association à un runtime ou nœud ProActive, lieu d'accueil de tous les objets actifs d'une même JVM. Dans le cas d'OSGi, ce nœud ProActive est démarré au moment de la livraison et activation de l'ensemble des bundles constituant la librairie ProActive. De même, le connecteur ProActive pour JMX est initialisé lors de l'activation du bundle par lequel son code est livré sur la passerelle OSGi.

Les méthodes que ce connecteur exporte sont inspirées de celles habituelles du MbeanServer, étendues avec une opération générique d'invocation d'opérations sur Mbeans compatible avec les contraintes liées à ProActive. En effet, pour qu'une invocation de méthode via ProActive s'exécute de manière effectivement asynchrone, il faut que le type de retour de la méthode soit un type extensible. C'est la condition qui permet de déclencher une opération

de manière asynchrone puisque un futur sera immédiatement rendu à l'appelant – ici l'outil d'administration – par la couche MOP de ProActive. Cette demande d'opération sera réifiée en tant que requête ProActive. Une fois parvenue dans la queue des requêtes de l'objet actif, elle sera servie de manière FIFO. Le service de la requête consiste à exécuter l'opération demandée de façon classique, via le MbeanServer. Si une réponse doit être rendue, elle sera remontée sous la forme d'une réponse ProActive. Du cotê de l'administrateur, il suffit de conserver une réference ProActive sur le connecteur pour pouvoir bénéficier de ce nouveau mode de communication avec la cible à adminsitrer.

Selon cette approche, réaliser des opérations d'administration sur un parc de machines de façon simultanée et homogène s'effectue aisément via un groupe de connecteurs. S'agissant d'un groupe d'objets actifs typés, les gains en terme de performance proviennent d'une part des optimisations réalisées lors de la préparation des requêtes, identiques et dont l'envoi pourra être multi-threadé, et d'autre part, par la potentialité d'exécution de ces opérations de manière effectivement parallèle. L'API de JMX ne permet évidemment pas l'accès distant à un groupe de Mbeans. Nous l'avons donc étendue dans ce sens (voir figure 4.1), en veillant à ne pas s'éloigner de la méthode usuelle pour l'accès et l'utilisation de Mbeans. Le programmeur peut donc créer un groupe (ProActive) de connections, puis, aussi simplement qu'il le ferait pour une connection, ouvrir cet ensemble de connections (en un seul appel de méthode), et ensuite, utiliser les méthodes d'accès au(x) Mbean(s) en fonction des interfaces de ces Mbeans. On peut remarquer que le fait que les connecteurs soient basés sur ProActive est transparent. L'introduction de ce concept de connexion asynchrone et de groupes de connexions a été pensée comme une extension naturelle de l'API JMX telle que la manipule le client Java JMX.

Des tests de performance réalisés grâce à un tout premier prototype démontraient un gain d'environ 50% dans le temps d'administration d'un parc de 10 machines. Au lieu de devoir attendre l'exécution de l'opération d'administration sur une passerelle donnée, avant de pouvoir passer à la passerelle suivante, l'administrateur attend seulement la réalisation du groupe d'opérations. Le délai d'attente perçu peut être de la sorte réduit de moitié si on compare à l'utilisation d'un connecteur RMI pour JMX.

```
//Création d'une connexion de groupe
JMXGroupConnexion connexion = new JMXGroupConnexion ();
//Connexion aux serveurs JMX
connexion.connect(new String[] {url1,…,url2});
//Récupération de la liste des bundles
ObjectWrapper
        bundleList = connexion.invoke(…"bundleList",…);
//Traitement spécifique du résultat …
Group gBundles = ProActiveGroup.getGroup(bundleList);
System.out.printl("…" + gBundles.get(i));
```

FIG. 4.1 – Exemple d'utilisation de l'extension de l'API JMX pour des groupes de connecteurs

### 4.2.5 Notifications JMX via ProActive

D'une manière symétrique, on veut que les notifications JMX soient envoyées par le MbeanServer de manière asynchrone. Pour cela, il suffit de rendre l'outil d'administration accessible à distance : cela se fait tout simplement en associant à cet outil un objet actif ProActive. Il faut aussi prendre soin de répertorier cet objet actif comme écouteur (listener) de ces événements JMX. Ici, l'objectif n'est pas de désynchroniser l'invocation et l'exécution de l'opération de réception de la notification, car cela ne présente guère d'intérêt en terme de gain de performance. Il s'agit par contre de mettre à profit les qualités additionnelles qu'offre ProActive par rapport à une communication RMI classique : communication de groupe, sécurisation des communications via des tunnels ssh, ou encore via l'authentification des partenaires d'une communication ProActive et l'encryption des données véhiculées par les requêtes et réponses [Section 2.2.2, page 139, section 4.4], maintien des liens de communication même en cas de migration des objets actifs cibles, etc.

On peut donc imaginer associer non pas une instance, mais un groupe d'instances de l'outil d'administration à une équipe d'administrateurs en charge d'un parc donné. Une notification peut ainsi être remontée au groupe d'instances. Comme une instance est construite sur la base d'un objet actif, elle peut migrer. Ceci peut s'avérer très pratique : un administrateur disposant sur son poste de travail d'une instance de l'outil d'administration peut ordonner la migration de cette instance (avec tout son état et contexte) sur un autre poste de travail. Cet autre poste de travail peut en particulier

être un équipement mobile : par exemple, son PDA qu'il emporte lorsqu'il doit quitter physiquement son lieu habituel de travail, alors qu'il doit continuer à superviser le parc. La fermeture propre et la réinstallation de l'interface graphique associée à l'outil peut être prise en charge par les méthodes `onDeparture` et `onArrival` de l'API ProActive pour la migration.

La figure 4.2 illustre via son interface graphique, les fonctionnalités qui pourraient être celles d'un outil d'administration de parcs de passerelles OSGi, tel que prototypé par nos soins du fait de notre participation au contrat PISE. Le parc est organisé en groupes, voire même en groupes de groupes. L'obtention de l'état de chaque passerelle peut être déclenchée via une invocation de méthode sur le Mbean présent sur chaque passerelle du groupe. Par exemple, on propose une méthode permettant d'obtenir la liste des bundles installés et leur état. Des événements remontent à l'outil sur changement dans la configuration de n'importe laquelle des passerelles, dès lors que ce changement génére un évenement OSGi.

### 4.2.6 Exécution transactionnelle de plan de déploiement

#### Problématique

Le passage à l'échelle des opérations d'administration est fortement lié à l'aptitude à automatiser les étapes de configuration des passerelles, tout en ayant l'assurance que celles-ci sont mises dans un état identique (état global cohérent en quelque sorte). Déployer une application sur une passerelle, et configurer en conséquence cette passerelle, peut nécessiter l'exécution d'une suite d'opérations (installation, désinstallation de bundles, activation, arrêt de services, etc), regroupées dans ce qui est couramment appelé un *plan de déploiement unitaire*. L'union des plans unitaires à exécuter sur les machines du parc ciblées est appelée *plan de déploiement global*. La difficulté dans l'exécution d'un plan unitaire, et par conséquent d'un plan global, est que rien ne garantit que l'exécution du plan arrivera à son terme sur toutes les passerelles cibles, même si un tel échec peut être considéré comme exceptionnel [54]. L'accès à certaines peut être momentanément coupé ce qui pourrait poser notamment un problème si l'exécution du plan est orchestrée pas à pas depuis l'outil d'administration. Même dans le cas contraire où chaque plan unitaire parviendrait à être délivré correctement à chaque passerelle cible et son exécution entièrement orchestrée localement, rien ne garantit que cette exécution termine avec succès. Il se peut par exemple que l'exécution locale

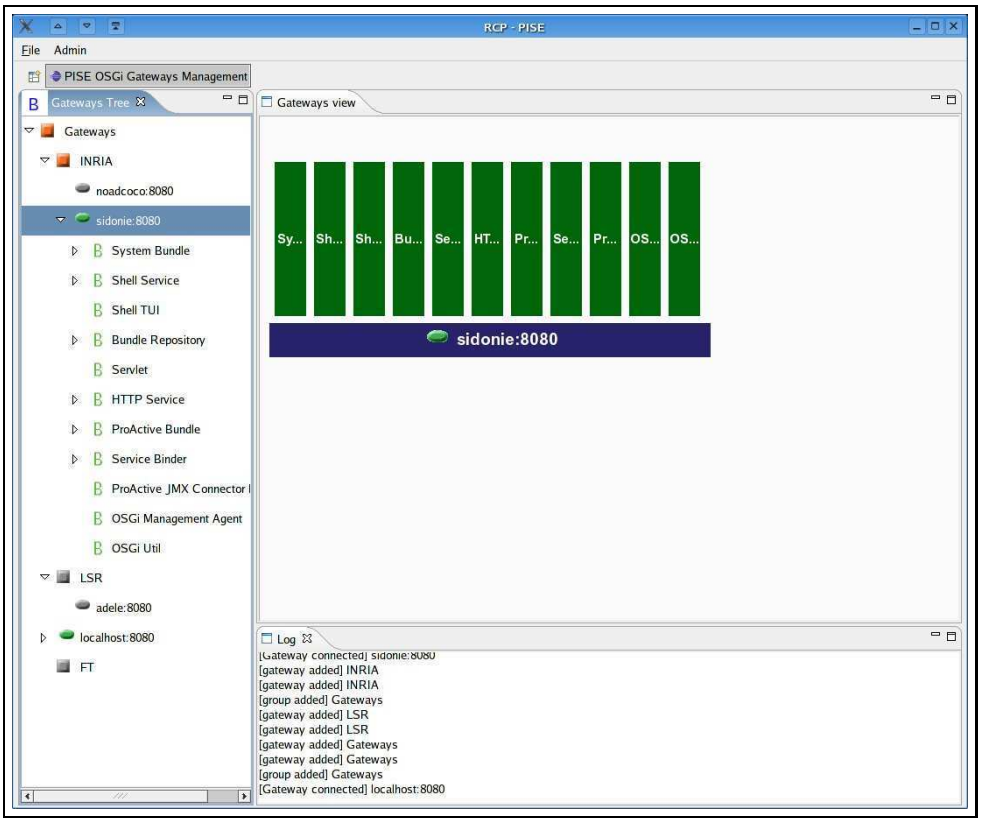



FIG. 4.2 – Interface graphique d'un outil d'administration d'un parc de passerelles OSGi

du plan nécessite de télécharger certains bundles. Or, rien ne garantit que ce soit possible à tout instant.

Au total, malgré le fait que le plan global ait été conçu en prenant en compte l'état de chaque passerelle de sorte à amener le parc dans un autre état, il se peut que certaines opérations du plan échouent. Ceci a pour effet de laisser le parc dans un état global incohérent, et aussi des passerelles dans un état intermédiaire peut-être non satisfaisant.

### Mode transactionnel

Il est donc utile de proposer un mécanisme de type transactionnel afin de pouvoir ramener le parc dans un état cohérent, stable, ce qui se traduit par une propriété de type atomicité (tout ou rien) sur le plan global, et par ricochet sur chaque passerelle concernée :

- soit le plan est exécuté dans son intégralité et enteriné par la validation de la transaction au niveau du plan global, et chaque passerelle reste donc dans son nouvel état considéré comme satisfaisant ;
- soit l'exécution du plan est annulée et par conséquent, l'état de chaque passerelle concernée doit revenir à ce qu'il était avant le début de l'exécution du plan. Le point de reprise considéré peut être celui qu'avait la passerelle antérieurement à l'exécution du plan ; d'autres points de reprise intermédiaires n'obligeant pas de tout défaire peuvent cependant avoir été indiqués dans le plan.

Il est certain que la décision, passerelle par passerelle en fonction de l'état réel dans lequel elle se trouve à présent, est à prendre idéalement par le commanditaire (en fonction de statégies [59]). En effet, c'est celui qui a calculé le plan qui a tous les éléments en main pour savoir comment respecter la cohérence du parc. Mais pour ce faire, il faut proposer à l'administrateur qui est l'exécuteur du plan, un certain nombre d'outils pour qu'il puisse contrôler finement l'exécution d'un plan et appliquer les stratégies demandées. En particulier, un de ces outils consiste à pouvoir défaire les actions que l'exécution d'un plan aurait réalisée, sur une passerelle, ou un groupe de passerelles. On est pourtant bien conscient que défaire tous les effets de bord que pourrait avoir eue l'exécution d'un plan peut être parfois impossible. Par exemple, si le plan a engendré l'installation de bundles ou activation de services, ceux-ci ont pu avoir des effets de bord, en influant l'exécution d'autres services déjà sur site, peut-être hors de la portée immédiate du plan. Ceci dit, dans une architecture à services dynamiques, ce n'est normalement pas un problème, puisqu'il est convenu que les services peuvent apparaitre et disparaitre dynamiquement [21].

**Transactions concurrentes** Les propriétés transactionnelles que l'on rencontre classiquement dans le domaine des bases de données, que sont atomicité, cohérence, isolation et durabilité ne sont pas vraiment celles requises [54], ou plutôt n'ont pas forcément la même interprétation. En particulier, on peut vraiment se poser la question de savoir si plus d'un plan est auto-

risé à s'exécuter à la fois. En effet, le calcul du plan de déploiement global part d'une vision globale à un instant donné, et en déduit le différentiel pour atteindre un nouvel état global du parc. L'entrelacement de l'exécution de plusieurs plans, associé à la possibilité d'annuler l'exécution de ces plans pose des problèmes de correction (ou cohérence) puisque l'isolation est impossible à assurer. L'exécution d'une action du plan de déploiement a un effet immédiat sur la passerelle, et génère ainsi des effets de bord sur son état, visibles aussi bien des applications déjà en place que de celles qui seraient déployées de manière concurrente.

> Malgré les problèmes de cohérence potentiellement engendrés, nous pensons qu'il est pourtant important de réfléchir à des solutions pour autoriser l'exécution concurrente de plans de déploiement. En effet, nous pensons que le modèle de passerelles à services ouvertes suppose leur approvisionnement, "re-approvisionnement", voire "desapprovisionnement" quasi constant en services, ceux-ci issus de différents fournisseurs.

De fait, ces fournisseurs ne se connaissent pas forcément et donc, ne se coordonnent pas forcément à l'avance avant de calculer et lancer un plan de déploiement global. Ainsi, nous posons comme condition que le calcul d'un plan de déploiement ne doit être possible qu'à partir d'un état global stable du parc. Ceci implique que pour calculer un tel plan, on doit attendre la terminaison de toutes les opérations de déploiement en cours sur le parc. Par contre, en partant d'un tel état global stable, nous autorisons le calcul et l'exécution du déploiement de plus d'une application, c'est-à-dire, la prise en charge de plus d'un plan global. Ceci permet à un fournisseur de services de planifier et exécuter le déploiement de plus d'une application une fois qu'il a la main sur le parc. Une motivation à cela est que l'exécution d'un déploiement global sur un parc de grande taille prend du temps. Si un fournisseur veut installer plus d'une application, ou service, sur ce parc, on peut espérer un gain en temps si ces installations peuvent s'exécuter de manière concurrente.

> Nous posons comme objectif la définition d'un mécanisme d'exécution de plan de déploiement global, qui soit parallèle (parce que ciblant un parc de machines de grande taille) et qui autorise sur un parc donné l'exécution concurrente de plus d'un plan de déploiement global à la fois.

Ce mécanisme pourrait utilement être intégré à des outils tels l'exécuteur de plan de déploiement de ORYA [59], ou le *Plan Manager* défini dans [47].

Dans le contexte de l'exécution concurrente de plans de déploiement globaux (chacun constitué d'un ensemble de plans de déploiement unitaires ciblant les passerelles du parc concernées), nous précisons à présent la sémantique des propriétés transactionnelles ACID, et décrivons comment la solution que nous avons définie et implantée permet de respecter cette sémantique.

### Atomicité

La propriété d'atomicité nécessite de savoir précisément quelles sont les actions requises permettant à chaque passerelle de repasser dans l'état initial qu'elle avait avant le début de l'exécution de la transaction (ou un état intermédiaire servant de point de reprise). Pour ceci, nous proposons que la description de chaque action définisse une opération *do()*, et son inverse *undo()* (typiquement, installer un bundle a comme inverse sa désinstallation). A chaque plan qui s'exécute sur une passerelle donnée, on associe, localement sur la passerelle, un log des actions demandées. On précise pour chacune si l'opération *do()* s'est faite. Ainsi, annuler, défaire, l'exécution d'un plan unitaire donné peut être réalisé sur chaque passerelle : cela consiste tout simplement à suivre le log, et à exécuter une opération *undo()* pour chaque opération *do()* indiquée.

Précisons qu'en plus de cela, chaque opération ayant un effet sur l'état d'une passerelle se traduit par l'occurence d'un événement OSGi, remonté par une notification JMX vers l'outil d'administration. Ainsi, faire ou défaire l'exécution d'un plan unitaire donné qui a eu pour effet de modifier l'état de la passerelle OSGi associée peut en dernier recours être piloté depuis l'outil d'administration. Il est ainsi possible de se baser sur l'hypothèse que des états globaux du parc existent en base de données, et que par rapport au dernier état global connu, le nouvel état global est l'incrément tel que perçu via l'ensemble des notifications reçues qui reflétent les changements d'états ayant eu lieu sur chaque élément du parc. Bien sûr, il se pose la question de l'éventuelle perte ou délai d'acheminement de ces notifications en cas de déconnexion temporaire. Au contraire, un mécanisme local à chaque passerelle est plus fiable pour savoir exactement quels changements d'état ont bien eu lieu. Mais, les deux mécanismes peuvent utilement être combinés.

### Cohérence

La propriété de cohérence signifie dans ce contexte [54], que les applications (ou services) qui fonctionnaient correctement avant l'exécution du plan, continuent à le faire de la même manière ; et de même, que l'application installée fonctionne correctement. Assurer ces deux points relève de la phase du calcul du plan : effectivement, les choix d'installation et d'activation de briques logicielles constituant une application (ou un ensemble de nouvelles applications) doivent prendre en compte les contraintes, besoins, pre-requis qu'auraient d'autres applications déjà sur site (ou faisant partie du même ensemble). Par ailleurs, le calcul du déploiement d'une application donnée peut préciser des contraintes à respecter pour avoir un comportement jugé correct de l'application. En conséquence, on peut aussi en déduire les conditions d'annulation complète ou partielle de l'exécution du plan, de sorte à maintenir l'application dans un état jugé acceptable pour son bon fonctionnement. En particulier, certaines applications peuvent être constituées de modules dits obligatoires, et d'autres optionnels. L'échec de l'exécution du plan concernant le déploiement de modules optionnels n'est peut-être pas critique pour un fonctionnement correct de l'application.

### Isolation

L'isolation est probablement la propriété la plus difficile, voire celle impossible à assurer dans le cadre du déploiement. En effet, une fois des bundle(s) installé(s) et le(s) service(s) correspondant activé(s), les effets sur d'autres services présents mais pourtant non concernés par le plan sont immédiats même si la transaction n'est pas encore validée, et donc impossibles à annuler (par exemple, permet la poursuite de l'exécution d'un service qui jusqu'à présent était bloqué).

L'isolation est également difficile à assurer si on considère l'exécution concurrente de deux ou plusieurs plans : l'exécution de n'importe lequel des plans peut avoir des effets de bord sur l'exécution de l'autre. Un cas simple mais suffisamment illustratif est le suivant : supposons que chacun des 2 plans exprime le besoin d'avoir un bundle donné (et similaire) installé sur une passerelle donnée. Comme ce bundle n'était pas présent lors du calcul du plan, l'action d'installation est inscrite *dans chacun des plans unitaires* correspondants. Selon l'entrelacement de leur exécution sur la passerelle, le bundle en question sera déjà présent du point de vue du déroulement de l'exécution

d'un des 2 plans. Le seul moyen simple d'isolation serait de verouiller l'accès à la passerelle dès qu'elle commence à exécuter un plan de déploiement, mais cela impliquerait la sérialisation de l'exécution de plusieurs plans, et non la concurrence de ces exécutions !

Si l'on veut éviter un tel verrouillage et autoriser l'exécution concurrente de plans de déploiement (à condition bien sûr que ceux-ci soient compatibles), on propose un mécanisme qui nous semble simple et pourtant efficace. Dans le log de l'exécution de chaque plan de déploiement unitaire, nous proposons de conserver toutes les actions de ce plan, en distinguant celles qui ont effectivement été faites (il leur correspond des opérations *do()*), de celles qu'il n'a pas été nécessaire de faire, car elles avaient déjà été faites concurrement par l'exécution d'autres plans. Ces autres actions sont regroupées dans ce que nous appelons un *plan de compensation*. Ce plan compensatoire est dynamiquement construit et associé à chaque log. Nous proposons d'associer à l'annulation d'une transaction et donc à l'annulation de l'exécution du plan unitaire associé sur une passerelle donnée, l'exécution des plans de compensation correspondants aux transactions qui ont été concurrentes à celle en cours d'annulation, sur cette même passerelle. Nous fournissons donc un mécanisme qui automatiquement sur annulation d'une transaction, est capable d'orchestrer l'exécution des plans compensatoires nécessaires sur chacune des passerelles concernées.

**Exemple** Supposons que le plan unitaire en cours d'annulation (p1) est celui qui avait installé un bundle que d'autres plans unitaires concurrents à p1 (p2, p3, ...) auraient aussi voulu installer. Alors, p2, p3, ...qui ont besoin de ce bundle doivent à présent reprendre leur déroulement pour tenter de l'installer. La première installation réussie de ce bundle s'inscrit évidemment dans le log associé au plan qui en est à l'origine (l'indication associée à cette action du plan est cette fois-ci l'opération *do()*). A nouveau, les autres plans qui tentent aussi d'exécuter cette même action vont donc mémoriser le fait qu'ils ne l'ont pas eux-mêmes réalisée, mais que l'exécution d'un autre plan l'a faite pour eux. Cette action apparait donc dans leurs plans compensatoires respectifs.

Si on pousse l'idée un peu plus loin, on s'aperçoit vite que l'exécution de tels plans de compensation est aussi nécessaire concernant des transactions avec lesquelles il y a eu concurrence d'exécution, même si celles-ci sont depuis validées. En effet, dans une transaction validée suite à la réussite de

l'exécution du plan global, certaines des actions demandées n'auront pas eu à s'exécuter vraiment, puisque réalisées par d'autres exécutions concurrentes. Si une de ces exécutions concurrentes est finalement annulée et défait donc ce qu'elle a fait, cela peut avoir également un effet sur une transaction pourtant déjà validée.

Nous cherchons donc à tout prix à éviter des annulations en cascade concernant des transactions concurrentes, d'autant plus si certaines ont déjà été validées. Pour cela, l'idée est de conserver trace pour toute exécution de plan unitaire, du plan compensatoire associé, et ce même après la validation de la transaction correspondante. Remarquons que la présence de ce plan compensatoire reste nécessaire tout au long de la vie de l'application. Cela tant qu'il contient encore une liste d'actions supposées avoir été exécutées selon ce plan, alors qu'en fait elles avaient été exécutées concurrement selon d'autres plans. Cependant, ce plan compensatoire est voué à raccourcir et donc, à disparaitre avec le temps.

#### Durabilité

La propriété la plus simple à respecter est par contre celle de la durabilité puisque les actions initiées par le plan ont des effets réels et donc pérennes si l'exécution n'est pas annulée.

### 4.2.7 Exécution parallèle de plan de déploiement

Un plan de déploiement global concerne un ensemble de passerelles. Plus précisément, à chaque passerelle cible peut correspondre un élément du plan global, c'est-à-dire un plan de déploiement unitaire. Rappelons que le déclenchement de l'exécution d'un déploiement unitaire se fait en invoquant une méthode asynchrone sur le Mbean de chaque passerelle avec comme paramètre le plan unitaire qu'elle doit exécuter. Rappelons aussi que les connecteurs JMX-ProActive des passerelles concernées peuvent être regroupés dans un groupe ProActive. Du coup, le déclenchement et l'exécution de ces plans unitaires est parallèle en passant par ce groupe.

On peut utiliser le mode `scatter` (voir [Section 2.2.3, page 171+3, section III.A]) concernant les paramètres d'appel d'une méthode sur un groupe, afin de distribuer à chaque passerelle du groupe le plan unitaire qui lui correspond. Le plan global est donc vu comme un groupe de type scatter. Il est découpé selon une relation 1-1 en fonction du groupe de passerelles ciblées.

Ceci nécessite au préalable, d'avoir organisé de manière similaire le groupe de connecteurs et le groupe de plans unitaires. Ceci peut par exemple être la première opération à effectuer dans l'outil qui se charge de l'exécution d'un plan de déploiement.

Un cas particulier plus simple, est celui où le plan global contient un seul plan unitaire. Ce plan unitaire est le même pour chaque passerelle ciblée. Il suffit alors de considérer le plan global comme un paramètre d'invocation de méthode, de type `broadcast` (qui est le type par défaut).

Si le parc ciblé par le plan global est très grand, le point (l'outil d'administration) depuis lequel se déclenche son exécution risque de constituer un goulot d'étranglement. Dans ce cas, nous avons deux possibilités : nous pouvons créer des groupes hiérarchiques de groupes de connecteurs ; nous pouvons aussi créer des sortes de copies miroirs de l'objet actif associé à l'outil d'administration qui pilote l'exécution du plan global. Chaque miroir a en charge un sous-groupe de passerelles. Le nombre de tels miroirs à créer, éventuellement dynamiquement, dépend du degré de parallélisme et de répartition visé. Il peut être le résultat d'un partage équilibré du nombre de passerelles ciblées. Mais, il peut aussi tenir compte de contraintes liées à l'organisation ou aux performances des réseaux impliqués. Quand le découpage du groupe de passerelles est effectué, il faut aussi réaliser la réorganisation du plan global en groupes de groupes. Ce qui est encore du ressort de l'outil d'exécution du plan, et non de l'outil de calcul du plan.

Des travaux restent à mener en ce qui concerne l'évaluation des gains effectifs en terme de performance d'un déploiement global lorsqu'on applique de telles parallélisations ; de même, lorsque on exécute concurrement plusieurs déploiements globaux.

## 4.3 Conclusion

Nous avons illustré comment un paradigme de programmation à objets mobiles, parallèles pouvait trouver son application dans un tout autre domaine que le calcul parallèle haute performance : dans le domaine de l'administration d'infrastructures matérielles ou logicielles, elles-mêmes largement réparties et de grande taille. Nous pensons vraiment que l'utilisation d'outils inspirés ainsi de concepts issus du parallélisme et de la répartition se justifiera de plus en plus, du fait de l'omniprésence que prend l'informatique, et en particulier l'informatique communicante et mobile. De tels outils sont tout

particulièrement nécessaires pour l'approvisionnement et le suivi d'applicatifs (ou services) sur de tels supports.

# Chapitre 5

# Conclusion - Perspectives

Le bilan des travaux déjà réalisés et des résultats obtenus permet d'envisager de poursuivre les recherches selon 4 axes complémentaires. L'objectif reste la maitrise de plateformes réparties à large échelle, incluant évidemment les grilles de calcul : que ce soit du point de vue de la programmation d'applications pouvant tirer avantage de telles infrastructures, comme de celui de leur mise en œuvre effective (déploiement, portabilité, supervision). Les recherches pour atteindre cet objectif pourraient s'articuler ainsi :

1. Programmation parallèle orientée objet
2. Composants répartis et parallèles
3. Interopérabilité des intergiciels de grille
4. Déploiement et supervision

L'ensemble devrait pouvoir contribuer à la réalisation d'un ambitieux projet collectif : celui de la définition d'un modèle de composants pour la grille *Grid Component Model (GCM)* entrepris par le consortium CoreGRID [R16].

## 5.1 Programmation parallèle orientée objet

### 5.1.1 Modèle OO-SPMD

Comme exposé dans le chapitre 3, nous avons proposé un paradigme de programmation parallèle SPMD (Single Program Multiple Data), original dans le sens où il se fonde sur des groupes typés d'objets représentant les activités parallèles. Il est à comparer avec d'autres plus classiques, tels ceux

fondés sur l'échange explicite de messages entre processus impératifs, comme prôné dans la norme MPI (Message Passing Interface). Notre paradigme offre a priori plus de souplesse, de flexibilité pour le programmeur (de par le fait que l'ordre de traitement des messages est plus paramétrable ; de par le fait qu'il permet de réutiliser du code, et en même temps de l'étendre pour le traitement de nouveaux types de messages). Il serait nécessaire de poursuivre des expérimentations concernant les qualités escomptées de ce modèle, ce sur des applications d'envergure réaliste, du type de ce qui a déjè été entrepris, en collaboration avec l'équipe CAIMAN de l'INRIA concernant un code de simulation d'ondes électromagnétiques [C13]).

### 5.1.2 Gestion des défaillances au niveau applicatif

Ce mécanisme de groupes d'objets typés permet aussi de construire des groupes d'activités hiérarchiques (et ainsi de structurer l'application de manière à mieux s'adapter à la topologie de l'architecture la plus courante pour des grilles de machines, qui est multi-grappes). Ce mécanisme de groupes devrait permettre de programmer des stratégies, éventuelllement sophistiquées, de gestion de défaillances d'activités qui sont courantes dans des environnements à très grande échelle. Nous cherchons à illuster ce potentiel dans le cadre d'une récente collaboration entre mathématiciens et informaticiens, en l'occurence l'équipe OMEGA de l'INRIA et Supelec. Il s'agit de concevoir une infrastructure logicielle, tirant profit de la formidable puissance de calcul offerte par les grilles, afin de fixer, de manière rapide et statistiquement bonne, le prix de produits dérivés dans le domaine de la finance ou de l'assurance (dans notre cas, options d'achat ou de vente, de type européen ou américain, fondées sur des paniers d'actifs – par exemple actions – sous-jacents, incluant ou non des variations de type barrière, ...) [C21]. Pour le moment, l'architecture logicielle proposée se fonde sur l'usage de groupes d'objets ProActive, suivant un pattern maitre(s) / esclaves (voir figure 5.1). Les stratégies de répartition du travail et de la gestion des défaillances sont par contre explicites et relativement mélangées au sein du code fonctionnel.

Une perspective est d'étudier une alternative à la résolution de ce type de problèmes par une approche objets. Il s'agit d'adopter une approche de type squelette, donc paramétrable, bâtie autour de composants Fractal-ProActive ou GCM comme suggéré dans [B3], [14], ...Les squelettes seraient bâtis comme des assemblages de composants génériques. La fonction de gestion des défaillances ou celle de répartition de travail devrait ainsi pouvoir être

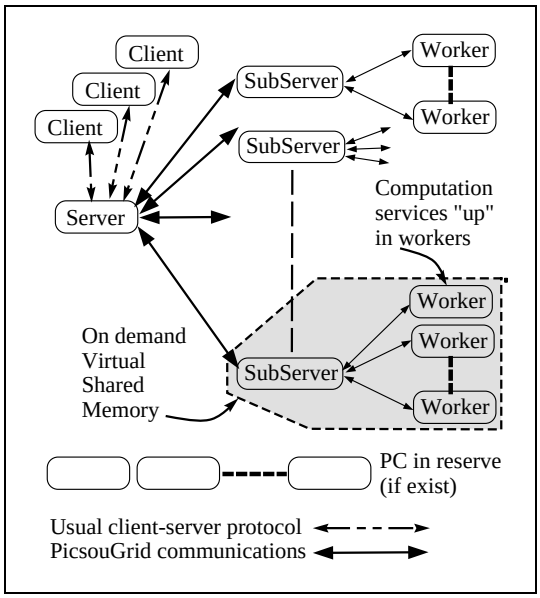



FIG. 5.1 – Architecture logicielle par groupes sur plusieurs niveaux pour le support d'applications de type maitre-esclave - Utilisation dans le cas de calculs en finance

conçue comme un contrôleur au niveau du composite qui représente le squelette. Cela permettrait de mieux séparer le code applicatif de celui lié à la gestion de problématiques de niveau non-fonctionnel.

## 5.2 Composants répartis et parallèles

Cet axe constitue la poursuite des travaux qui étudient l'adéquation d'une approche par composants logiciels dans la fourniture d'applications réparties et parallèles, comme exposé dans le Chapitre 3, section 3.3.

Nous pensons qu'il est important de poursuivre l'étude des problématiques spécifiques liées au contexte d'exécution sur des grilles de calcul, et qui en fait dépassent souvent ce contexte.

### 5.2.1 Raccourcis dynamiques et Reconfiguration

Pour palier à la latence importante dans les communications, nous avons vu qu'il est nécessaire de prévoir des modes de communication asynchrones, et optimisés, par introduction dynamique de raccourcis court-circuitant les membranes des composites. Une étude plus poussée, entamée par Ludovic Henrio, se justifie afin de proposer un protocole de reconfiguration correct en présence de telles optimisations. Il est nécessaire de stopper un composant pour le reconfigurer. Mais un composant composite étant réparti, il est délicat de stopper de manière atomique toute son activité, qui inclut des réceptions de messages via les raccourcis existants. Du coup, il ne semble pas évident de pouvoir conserver intact l'ordre des messages traités par un composite en présence de reconfiguration. Autrement dit, il faut parvenir à définir un protocole qui assure le même ordre de traitement de l'ensemble des messages que l'on utilise ou non l'optimisation.

### 5.2.2 Modèle OO-SPMD et composants hiérarchiques parallèles

Les composants regroupant des activités parallèles devraient pouvoir profiter d'optimisations dans l'invocation de services offerts et requis, grâce aux interfaces de type multicast ou gathercast. Des travaux inspirés des solutions connues pour le MxN restent nécessaires. Ils permettront de combiner les 2 types d'interfaces afin d'exprimer et réaliser plus efficacement le couplage de 2 composants parallèles hiérarchiques comprenant des nombres quelconques de réplicas.

La collaboration avec des numériciens et d'autres informaticiens, dans le cadre du projet DiscoGRID de l'ANR s'est donné l'objectif suivant : investiguer l'approche par composants hiérarchiques pour tirer partie de l'organisation physique des grilles, habituellement multi-clusters de PCs, eux-mêmes multi-processeurs. L'utilisation combinée de groupes d'objets actifs organisés selon le modèle OO-SPMD, et de composants contenus dans un composant hiérarchique parallèle pourrait permettre d'atteindre cet objectif. En effet, le compostant hiérarchique parallèle servirait à structurer et traiter les communications inter-clusters engendrées par les codes parallèles patrimoniaux, chacun d'eux s'exécutant uniquement au sein d'un même cluster. Un code parallèle serait emballé grâce à composant primitif basé sur un objet actif, ce composant constituant un des composants gérés par le composant

parallèle hiérarchique englobant. Les interactions entre ces codes parallèles numériques sont riches : typiquement, elles suivent les schémas d'interaction globale usuels 1-N, N-1, N-N. Il semble alors naturel d'essayer d'appliquer l'approche OO-SPMD aux objets actifs emballant les codes MPI, pour faciliter le support de tels schémas globaux.

### 5.2.3 Comportements autonomes

Un des sujets de recherche les plus actifs actuellement tourne autour de la problématique des comportements autonomes, face à toutes sortes de situations : face aux pannes (self-healing, self-repairing), à la dégradation des performances (self-optimizing), aux risques d'attaques (self-securing), aux besoins de supervision, d'administration et de configuration sans intervention humaine (self-monitoring, self-configuring, etc), ....

Ce besoin dépasse bien largement les fédérations de ressources de calcul telles les grilles, et s'exprime aussi dans le domaine du pair-à-pair au sens large, dans le domaine des serveurs d'applications répliqués sur clusters (auquel notre participation à l'ARC INRIA AutoMan nous sensibilise), délocalisés en partie sur des serveurs dits *Edge servers* placés par les ISPs aux frontières de l'Internet au plus près des usagers [27], ou encore pour les réseaux de communication locaux mobiles. Notre participation au projet européen Bionets nous permet d'être sensibilisés à une démarche auto-organisationnelle inspirée par exemple des processus biologiques ou biochimiques, des comportements collectifs que l'on peut observer chez les insectes sociaux, .... L'objectif du projet est de tenter d'appliquer ces processus à l'auto-organisation des réseaux mobiles en mode ad-hoc.

Une approche par composants logiciels devrait bien se prêter à la prise en compte de telles propriétés d'autonomie (*self-* properties*), du fait d'une séparation assez nette entre aspects fonctionnels et non fonctionnels. Par ailleurs, les comportements autonomes qui doivent piloter les parties fonctionnelles ou non fonctionnelles des composants peuvent être complexes : ils peuvent avoir un impact sur le cycle de vie, les liaisons entre composants ; ils peuvent requérir l'acquisition d'informations résultant de l'observation du contexte d'exécution, et il se peut qu'eux-mêmes évoluent.

Une possibilité pour avoir un système global de composants qui soit autonome consiste à procéder à l'introspection de ce système. Lorsque le système de composants résulte d'une description des assemblages de composants, stockée dans un fichier ADL (Architecture Description Language), cet ADL

et les opérations d'introspection peuvent se complémenter. Le but est de créer une image miroir du système de composants [22]. En cas de problème, cette image permet de piloter la réparation, la réorganisation, ou l'évolution de l'assemblage des composants, guidées par des stratégies prédéfinies. C'est par exemple ce que permet le système Jade [13] pour l'auto-administration de serveurs J2EE répliqués (chaque duplica est emballé d'une façon sommaire dans un primitif Fractal ; l'image miroir du système est structurée comme un système composite Fractal et utilisée par le composant 'cerveau' qui orchestre les reconfigurations nécessaires pour réparer ou optimiser le système supervisé).

> Nous nous plaçons dans un cadre plus exigeant, où le degré de décentralisation est total. On ne veut faire absolument aucune hypothèse quant à l'existence d'un oracle qui en quelque sorte piloterait la réalisation des comportements autonomes. Il nous faut donc une solution par laquelle les comportements autonomes qui doivent piloter les composants se trouvent embarqués dans les composants eux-mêmes.

Ces comportements autonomes relèvent essentiellement des aspects non fonctionnels, donc dans le cas de Fractal, ils devraient pouvoir être présents dans la membrane des composants. Des travaux autour de Fractal, basés sur l'idée d'utiliser des composants pour programmer la membrane ont déjà débuté [71, 58, 26]. Nous proposons d'aller plus loin encore dans cette démarche, en permettant d'embarquer un système à composants Fractal tout à fait standard dans la membrane de chaque composant Fractal bâti sur ProActive [R16,R17,W22]. De fait, cette membrane peut aussi bien offrir ou requérir des services. Un composant équipé d'une telle membrane doit donc offrir des interfaces non fonctionnelles qui soient de véritables interfaces de type serveur ou client, potentiellement collectives, soit multicast ou gathercast. En effet, une prise de décision relevant de l'autonomie d'un composant hiérarchique peut nécessiter la participation coordonnée des composants contenus dans cette hiérarchie [6]. Eux-mêmes ont une membrane constituée d'un système à composants, et offrent ainsi des interfaces auxquelles se relier. Par ailleurs, une membrane doit pouvoir être hiérarchique, facilitant ainsi la maitrise de sa complexité potentielle, et de ses reconfigurations.

## 5.3 Interopérabilité des intergiciels de grille

Le concept d'une grille de calcul planétaire, telle que l'est l'Internet dans le cas d'un réseau de communication, n'est pas encore une réalité. Pourtant, c'est ce qui constitue le point de mire : pouvoir utiliser de la puissance de calcul en se branchant tout simplement sur la grille, sans se soucier d'où cette puissance provient. Même si les grilles déployées aujourd'hui sont de plus en plus de type inter-continental, elles sont utilisables en se pliant à des modes d'accès qui ne sont en général pas compatibles les uns avec les autres. Or, construire LA grille requiert un concensus technologique permettant d'interconnecter de manière transparente les différentes grilles déployées, pour que les applications puissent utiliser indifféremment telle ou telle portion de cette unique grille. L'absence actuelle de concensus est en train de se résorber. En effet, la communauté en a pris conscience et fournit des efforts en matière de standardisation des mécanismes d'accès aux ressources des différents types de grilles.

L'intérêt que nous portons à cette thématique concerne tout d'abord la manière de rendre inter-opérables des environnements d'exécution sur différentes grilles, afin de fournir une solution de portabilité pour les applications qui voudraient s'exécuter sur n'importe quelle grille, voire n'importe quelle combinaison hétérogène de grilles. Pour alimenter ces travaux, les collaborations établies entre l'équipe et l'European Telecommunication Standard Institute (ETSI) sont précieuses [W14, W20] : elles ont pour objectif de contribuer à la réflexion et l'établissement de standards dans ce domaine. Les standards sur lesquels un organisme comme l'ETSI se penche habituellement relèvent plus des aspects protocolaires de niveau réseau, utiles aussi pour l'émergence de grilles, que d'aspects programmation haut niveau. Nous pensons que des modèles de programmation de tels systèmes, comme une approche par composants, peuvent largement bénéficier de l'existence de standards visant les couches intermédiaires [W14,W20] (par exemple, publication et découverte de ressources ou de services techniques, déploiement des services sur ces ressources en s'inspirant des techniques issues de déploiement de composants [70]). Il n'empêche que certains points relevant de programmation répartie sur ce type de support méritent concensus et standardisation : composition hiérarchique, contrôle autonomique, services parallèles, etc, pourraient être de tels candidats. C'est donc également vers ce deuxième type de préoccupations qu'il nous semble utile d'agir, sachant que les industriels s'y investissent déjà beaucoup (voir par exemple les récentes proposi-

tions autour de SCA [11]).

De tels efforts de standardisation passent aussi par la mise en place de tests de conformance des mises en œuvre des spécifications de ces standards, et des tests d'interopérabilité de leurs différentes implémentations [R10, R14].

## 5.4 Déploiement et supervision

L'accroissement du degré de répartition, de la taille des parcs de machines et de services qu'elles hébergent rendent indispensable l'automatisation des opérations de déploiement puis de supervision. Les travaux entrepris et décrits dans le Chapitre 4 contribuent à cette automatisation.

Il pourraient être étendus au déploiement de systèmes à base de composants. Les infrastructures d'accueil peuvent être hétérogènes, nécessitant de livrer les composants selon des formats différents. Une approche de provisionnement générique et automatique pourrait sélectionner le format de livraison du composant en fonction du type du middleware d'accueil (par exemple, JVM standard ou JVM type OSGi, J2SE, J2EE ou J2ME). L'abstraction qu'offre les composants devrait permettre de masquer ces niveaux d'hétérogénéité même dans le cas de composants répartis. Ces travaux pourraient également être appliqués à la supervision de systèmes à base de composants. On peut facilement associer aux composants l'expression de leurs besoins en matière de services techniques de l'environnement d'accueil [17] (service de tolérance aux pannes, d'acquisition de ressources, ...). On pourrait étendre cette liste de besoins avec un service d'administration distante. Dans notre contexte, ce service serait mis en œuvre grâce au connecteur ProActive-JMX.

Les composants peuvent être exposés selon la technologie à services désirée. Ils pourraient donc être facilement branchés sur des bus de services, typiquement des ESBs (Enterprise Service Bus). On constate depuis quelque temps une forte convergence de l'intérêt porté au concept de services connectés au bus de l'entreprise [11, 42] et communiquant grâce à lui, et de l'intérêt pour le concept de grille d'entreprise, c'est-à-dire d'une fédération dynamique des ressources physiques au sein du réseau d'entreprise affectées à une application (notion de virtualisation de ressource).

Une approche combinée, où :

– les services sont mis en œuvre par des composants répartis et autonomes, capables d'interagir directement ou via un bus, ce bus lui-même

résultant d'un assemblage de ce type de composants,
- déployés sur les ressources choisies dynamiquement au sein du réseau d'entreprise selon leur disponibilité (self-provisioning)

nous semble constituer une bonne base pour supporter ce concept d'Enterprise Grid [43, 2].

Le démarrage d'une collaboration (projet AGOS : Architecture Grille Orientée Services) avec quelques uns des acteurs industriels dans ce domaine (Oracle, HP) devrait nous permettre de valider cette approche. Les résultats des collaborations initiées par le biais du projet RNRT PISE, du projet ITEA S4ALL (Services for All) et de l'ARC INRIA AutoMan devraient également y contribuer.

Finalement, la vision est celle d'une convergence forte entre les fédérations de machines organisées en grille, et les fédérations de services répartis exécutés sur ces machines. C'est celle d'une grille de services complètement globalisés dont la localisation importe peu : seule la qualité du service rendu est importante. L'ensemble des efforts de recherche menés dans la communauté peut contribuer à ce que cette qualité soit garantie. Cela passe par des outils performants permettant de configurer, déployer et administrer ces services ainsi que leur support [4, 29]. Cela passe aussi par des travaux autour de la gestion de l'autonomie.

Notre recherche et nos outils ont donc le potentiel de contribuer à l'émergence de nouveaux modes d'usage des grilles informatiques, cette fois-ci dans le monde de l'entreprise, et non plus uniquement dans le domaine scientifique. Les concepteurs de middlewares pour l'entreprise ont compris l'enjeu pour y arriver. Cela passe par la modularisation de leurs suites logicielles (serveurs d'entreprise : serveurs web, d'applications et de données), pour lesquelles la combinaison de l'approche par composants et provisionnement de code à la demande sur plateformes ouvertes semble adéquate [3].

# Deuxième partie

# Publications

# Chapitre 1

# Liste des articles

Articles supports de ce mémoire d'habilitation.

## 1.1 Programmation parallèle structurée

F. Baude and D. Skillicorn. Vers de la programmation parallèle structurée fondée sur la théorie des catégories. *Technique et science informatiques*, 13 :494–525, 1994.

## 1.2 Programmation à objets parallèle

### C++//

F. Baude, D. Caromel, and D. Sagnol. Distributed objects for parallel numerical applications. *Mathematical Modelling and Numerical Analysis Modelisation, special issue on Programming tools for Numerical Analysis, EDP Sciences, SMAI*, 36(5) :837–861, 2002.

### ProActive

L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid (chapter 9). Springer, 2006. Pages 205–229. ISBN : 1-85233-998-5.

L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *CCGrid 2005 : IEEE/ACM International Symposium on Cluster Computing and the Grid*, Pages 824–831, Vol. 2. April 2005.

F. Baude, D. Caromel, F. Huet, T. Kielmann, A. Merzky, and H.Bal. *Future Generation Grids*, chapter Grid Application Programming Environments. CoreGRID series. Springer, jan 2006. ISBN : 0-387-27935-0. Also as the CoreGrid TR003 report, `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0003.pdf`.

## 1.3 Programmation par composants

F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, LNCS, pages 1226–1242. Springer Verlag, 2003.

## 1.4 Administration

E. Reuter and F. Baude. System and Network Management Itineraries for Mobile Agents. In *4th International Workshop on Mobile Agents for Telecommunications Applications, MATA*, number 2521 in LNCS, pages 227–238. Springer-Verlag, 2002.

E. Reuter and F. Baude. A mobile-agent and SNMP based management platform built with the Java ProActive library. In *IEEE Workshop on IP Operations and Management (IPOM 2002)*, pages 140–145, Dallas, 2002. ISBN 0-7803-7658-7

# Chapitre 2

# Articles

## 2.1 Programmation parallèle structurée

### 2.1.1 Vers de la programmation parallèle structurée fondée sur la théorie des catégories.

[J1] F. Baude and D. Skillicorn. Vers de la programmation parallèle structurée fondée sur la théorie des catégories. *Technique et science informatiques*, 13 :494–525, 1994.

RECHERCHE

# Vers de la programmation parallèle structurée fondée sur la théorie des catégories

**François Baude* — David Skillicorn****

** 13S URA 1376 - Université de Nice Sophia-Antipolis*
*650 route des Colles, BP 145*
*06903 Sophia-Antipolis cedex*

*** Department of computing and information science*
*Queen's university*
*K7L 3N6 Kingston, Canada*

Ces recherches ont été financées par une bourse post-doctorale de l'INRIA pour M^me^ Baude, et pour M^r^ Skillicorn, par le *Natural sciences and engineering research council of Canada*

RÉSUMÉ. *Un modèle abstrait idéal pour le calcul parallèle devrait répondre aux besoins, à la fois en matière de génie logiciel et en matière d'efficacité des implantations. Nous montrons comment le style de programmation à parallélisme de données, restreignant le mode de calcul à un ensemble donné d'opérations calcul/communication bâties sur un certain moule, peut répondre à ces exigences. Cependant, il est nécessaire de remédier au choix souvent arbitraire des opérations du modèle. Nous y remédions en proposant une construction de types de données catégoriques.*

ABSTRACT. *An ideal abstract model for parallel computation should answer the needs of both software engineering and implementation efficiency. We show how the data-parallel style of programming, which restricts the computation mode to a given set of computation/communication operations built on a specific scheme, can fulfill these requirements. Nevertheless, data-parallel models provide operations which are usually arbitrarily chosen. We leave this restriction by providing a construction of categorical data-types.*

MOTS-CLÉS : *modèles de calcul parallèle, modèles abstraits, indépendance vis-à-vis des architectures, parallélisme de données, programmation fonctionnelle, types de données catégoriques, formalisme Bird-Merteens.*

KEY WORDS : *models of parallel computation, abstract models, architecture independance, data-parallelism, functional programming, categorical data-types, Bird-Merteens formalism.*

## 1. Introduction : Nécessité d'une approche intégrée pour le développement de logiciel parallèle

### 1.1. *Enoncé du problème*

Le défi auquel se trouve confrontée l'informatique parallèlle si elle veut enfin devenir un outil largement utilisé, même par les non spécialistes, est de proposer une méthode de développement de logiciels parallèles qui, tout en étant suffisament de haut niveau pour être indépendante des si nombreuses et différentes architectures parallèles, donne lieu cependant à des mises en œuvre efficaces.

En effet, la dépendance entre logiciel et architecture cible est souvent très forte, ceci dans un souci évident de recherche de performance (qui est, rappelons-le, le but essentiel du calcul parallèle). Mais, comme des nouvelles architectures parallèles sont développées chaque année, différant souvent assez fortement dans les détails et y compris dans leur nature, de lourds efforts peuvent être nécessaires pour adapter les logiciels existant à cette évolution des machines. Ceci explique pourquoi cette branche de l'informatique reste encore une (coûteuse !) affaire de spécialistes.

### 1.2. *Etat de l'art*

De nombreuses recherches tendant à établir un pont satisfaisant entre logiciel et architecture (mettant l'accent plus ou moins fortement sur l'un ou l'autre) ont déjà donné le jour à plusieurs modes de calcul parallèles. Citons par exemple :

**Les modèles BSP et LogP** (cf. [VAL 90a] et [CUL 93]) définissant des variantes, plus appropriées pour une mise en œuvre efficace sur une large gamme d'architectures, du modèle théorique standard d'algorithmique parallèle (le modèle PRAM [AKL 89]), mais qui du même coup sont plus complexes à manier. En effet, un algorithme qui soit à la fois indépendant des architectures et qui mène à une exécution la plus efficace possible sur une architecture spécifique est défini en considérant un jeu de paramètres (qui ne seront instanciés qu'à la compilation) capturant les caractéristiques essentielles des performances d'une architecture parallèle (essentiellement, latence et débit du réseau de communication, nombre de processeurs).

**La définition d'une norme de langage -généraliste- comme HPF** (High Performance Fortran) qui bien qu'en accord avec les exigences de la large communauté des utilisateurs numériciens, et bien qu'exploitant le parallélisme de données (concept qui se révèle être incontournable pour l'obtention de la propriété d'indépendance vis-à-vis des architectures parallèles [BAU 91]), fait largement appel aux utilisateurs pour résoudre le placement et la distribution des données du programme.

**A l'opposé, on trouve des modèles dont le souci primordial est le "bien-être" des programmeurs,** donc d'un niveau d'abstraction élevé, cachant le plus de points possibles liés à l'implantation (par exemple Linda ou Unity [CHA 88]). Mais, une fois un

programme obtenu, il est difficile de prédire ses performances sur une architecture particulière, car la distance à franchir jusqu'à l'implantation est trop grande donc l'espace des solutions trop vaste.

**L'abandon du développement de logiciel parallèle par le biais d'un langage généraliste, au profit d'un modèle imposant une restriction sur les formes possibles de programmes.** L'utilisateur dispose d'un ensemble de squelettes comme dans les approches de [DAR 93], [COL 89] ou d'un ensemble de primitives à parallélisme de données (cf. la suite de l'article, ou le modèle vector-scan [BLE 90]). Les formes de ces squelettes ou primitives doivent être suffisamment peu nombreuses pour que leur implantation la plus efficace possible sur différentes architectures ne représente pas un travail colossal. Par ailleurs, ceci permet de remonter au niveau de l'utilisateur une estimation de performance pour chaque forme, voire éventuellement pour chaque type d'architecture dans le cas où les performances ne seraient pas invariantes d'une architecture à l'autre. L'acceptation de ces approches restrictives dépend alors de l'étendue des instanciations possibles des formes implantées [1], mais également de l'existence d'une méthodologie permettant à l'utilisateur d'être aidé dans le processus consistant à exprimer un calcul quelconque en termes des squelettes ou primitives autorisés.

Notre solution appartient à cette dernière classe de modes de programmation parallèle.

### 1.3. *Propriétés recherchées*

Ainsi, un modèle idéal de programmation parallèle est "tiraillé" entre deux ensembles de contraintes que nous allons maintenant plus clairement distinguer, d'une part celles relatives au développement du logiciel qui puisse avoir une longue durée de vie, d'autre part celles concernant son implantation efficace sur les architectures parallèles présentes voire futures.

**Indépendance vis-à-vis des architectures** Le modèle ne doit pas inclure de concepts qui soient propres à un type particulier d'architectures (par exemple, à passage de messages explicite, ou quelque hypothèse que ce soit sur l'organisation de la mémoire). En d'autres termes, il doit être possible de faire s'exécuter un même programme sans modification sur toute plate-forme.

**Simplicité intellectuelle** Le degré de parallélisme pouvant être massif, le programmeur ne doit raisonnablement pas avoir à se soucier des points relatifs à la synchronisation, la communication entre ces nombreuses tâches ainsi que leur multiplexage sur les différents processeurs.

**Méthode de développement** Il doit exister une méthodologie pour le développement du logiciel permettant de construire des programmes complexes qui soient corrects. Bien qu'une approche fondée sur la vérification de programmes puisse être envisageable, mais

[1] dans la plupart des langages à parallélisme de données l'ensemble des opérateurs est pré-défini (modèle vecteur-scan [BLE 90], POMPC [PAR 92], etc) et ne s'applique qu'à un type de donnée parallèle (liste en CM-Lisp [STE 86], tableau dans les Fortran parallèles, multi-ensemble en Gamma [BAN 91], etc.)

relativement complexe, des approches par raffinements successifs semblent être mieux appropriées. Cela nécessite que le modèle possède une solide sémantique.

Les propriétés énoncées peuvent être vérifiées en rendant le modèle suffisamment abstrait. Cependant, ce serait au détriment des propriétés qui suivent et prôneraient un modèle plus concret.

**Existence de mesures de coût** Il faut un moyen pour estimer les performances (ou coûts en termes de ressources) qu'aura un programme, même avant qu'il ne soit forcément terminé. Autrement, il est difficile de développer du logiciel par dérivation, puisque le choix dans un souci d'efficacité, entre diverses manières de résoudre le problème ne pourrait être fondé.

**Implantation (théorique) efficace** Le modèle doit pouvoir être efficacement implantable sur une large gamme d'architectures. A ce titre, une implantation la plus efficace possible (c'est à dire optimale) est définie comme telle que le produit de la complexité d'une implantation en temps par le nombre de processeurs est asymptotiquement du même ordre que son équivalent sur une machine théorique de référence pour le calcul parallèle (PRAMs ou circuits booléens). C'est une contrainte très forte, mais qui si elle peut être respectée, assure que le modèle minimise les coûts intrinsèques du parallélisme et de la distribution.

Notre solution (à ces cinq points) est la construction de types de données catégoriques dans l'esprit des langages munis de primitives à parallélisme de données, mais telle que le modèle permet de disposer d'un large spectre de types structurés.

## 2. Types de données catégoriques

Les principes de la construction (dus à Malcolm [MAL 90]) vont être énoncés et appliqués à un exemple simple classique, celui des listes. Cette construction a également pu être appliquée aux ensembles et multi-ensembles [BAN 93], aux arbres [GIB 91], aux tableaux [BAN 92], aux graphes [SKI 93a]. Nous discuterons dans la section 3 des propriétés de cette construction dans le cadre de la programmation parallèle.

Nous supposons que le lecteur possède les bases élémentaires de la théorie des catégories ([LAM 86], [PIE 91]).

Le point de départ est une catégorie dont les objets sont des ensembles ou des types de base (l'ensemble des entiers, des booléens, ...) et dont les flèches sont des fonctions totalement calculables entre eux. Bien que la construction se révèlera être polymorphique, commençons par raisonner à partir d'un objet particulier, $A$ de cette catégorie (par exemple, les entiers naturels $\mathbb{N}$) et construisons un type de donnée catégorique (TDC) basé sur $A$ (ce sera par exemple, le type liste d'entiers).

La construction se réalise selon les étapes suivantes :

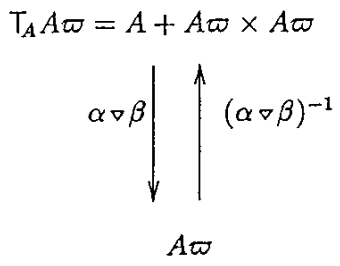



**Figure 1.** *Construction du type catégorique*

### 2.1. *Définition de constructeurs*

On choisit des constructeurs pour le TDC qui indiquent comment construire un objet du nouveau type noté $A\varpi$ à partir des différentes pièces qui le composent. Typiquement,

$$\alpha : A \rightarrow A\varpi$$
$$\beta : A\varpi \times A\varpi \rightarrow A\varpi.$$

Le premier constructeur montre comment obtenir à partir d'un objet de type $A$ un objet "singleton" du type construit $A\varpi$, alors que le second indique comment utiliser 2 objets du type $A\varpi$ pour obtenir un objet plus gros. Le type liste d'entiers (noté $\mathbb{N}*$) a 3 constructeurs :

$$\begin{array}{lcl} [\,] & : & unit \rightarrow \mathbb{N}* \\ [\cdot] & : & \mathbb{N} \rightarrow \mathbb{N}* \\ \bowtie & : & \mathbb{N}* \times \mathbb{N}* \rightarrow \mathbb{N}* \end{array}$$

où [ ] construit la liste vide, [·] la liste singleton, et $\bowtie$ par concaténation, un objet plus gros à partir de 2 listes de $\mathbb{N}*$. $\bowtie$ doit être spécifié comme une loi associative, dont l'élément neutre est la liste vide [ ], afin de reproduire l'ordre linéaire d'une liste (sinon cette construction représente celle de l'arbre binaire).

Dans une catégorie munie de coproduits, il y a une flèche de jonction $\alpha \triangledown \beta : A + A\varpi \times A\varpi \rightarrow A\varpi$ (sur l'exemple, $[\,] \triangledown [\cdot] \triangledown \bowtie$: $unit + \mathbb{N} + \mathbb{N}* \times \mathbb{N}* \rightarrow \mathbb{N}*$). Cette flèche de jonction a bien un rôle de constructeur, à condition que les différentes pièces que l'on connecte existent dans la catégorie. Pour qu'elles existent, nous utilisons un foncteur *polynômial*, noté T. T appliqué à un objet du type construit le décompose en ses différentes pièces (l'effet est d'« inverser »la flèche $\alpha \triangledown \beta$). Dans le cas général, $T = K_A + Id \times Id$ où $K_A$ est le foncteur constant pour le type $A$, donc $TA\varpi = A + A\varpi \times A\varpi$. T dépendant de l'objet original $A$, on le note $T_A$. Comme T est polynômial, il a un point fixe dans la catégorie, et c'est justement ce point fixe que nous posons comme étant le TDC $A\varpi$ (cf. figure 1). Nous avons donc un isomorphisme entre $T_A A\varpi$ et $A\varpi$.

Pour $\mathbb{N}*$, $T_1 = K_{unit}$, $T_2 = K_{\mathbb{N}}$, $T_3 = \mathrm{Id} \times \mathrm{Id}$ et $T_{\mathbb{N}} = T_1 + T_2 + T_3$.

### 2.2. *Les T-algèbres*

Ce foncteur T capture donc précisément la manière dont les pièces sont connectées pour constituer le nouveau type. Il peut également être appliqué à d'autres types pour donner des objets qui sont de même nature que le nouveau type.

On peut donc définir la $T_A$−algèbre $T_A A\varpi \rightarrow A\varpi$ (resp. $T_{\mathbf{N}}\mathbf{N}* \rightarrow \mathbf{N}*$, qui représente le monoïde libre de listes basé sur $\mathbf{N}$) et qui va elle-même être un objet dans une nouvelle catégorie, celle des $T_A$−algèbres (que l'on notera $T_A$-Alg).

De plus, tout monoïde qui préserve la structure de $A\varpi$, va être représenté par un objet de $T_A$-Alg, une $T_A$−algèbre. En effet, à toute fonction $f = f_1 \triangledown f_2 \triangledown f_3$ où

$$f_1 : unit \rightarrow X$$
$$f_2 : A \rightarrow X$$
$$f_3 : X \times X \rightarrow X$$

correspond la $T_A$−algèbre $T_A X \rightarrow X$, où $f_1$ définit l'élément de l'algèbre qui est l'élément identité pour l'opération associative $f_3$.

Par exemple, somme : $\eta \triangledown id \triangledown +$, où

$$\begin{array}{lcl} \eta & : & unit \rightarrow 0 \\ id & : & \mathbf{N} \rightarrow \mathbf{N} \\ + & : & \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}, \end{array}$$

représente une $T_{\mathbf{N}}$−algèbre, plus précisément, le monoïde $(\mathbf{N}, +, 0)$. Nous pouvons alors illustrer l'homomorphisme, que nous pourrions appeller $cumul : (\mathbf{N}*, \bowtie, [\,]) \rightarrow (\mathbf{N}, +, 0)$, tel que $cumul(x \bowtie y) = cumul(x) + cumul(y)$ où $x$ et $y$ sont des listes d'entiers, qui se traduit donc dans $T_{\mathbf{N}}$-Alg par une flèche entre l'algèbre $(T_{\mathbf{N}}\mathbf{N}* \xrightarrow{[\,] \triangledown [\cdot] \triangledown \bowtie} \mathbf{N}*)$ et l'algèbre $(T_{\mathbf{N}}\mathbf{N} \xrightarrow{\eta \triangledown id \triangledown +} \mathbf{N})$. Les flèches qui représentent des homomorphismes entre algèbres sont appelées *catamorphismes* dans la catégorie des T-algèbres, et pour distinguer ces flèches de $T_A$-Alg, on se sert de la notation $(\!|\ \ldots\ |\!)$. Un catamorphisme a la particularité d'établir une correspondance univoque entre deux algèbres. C'est pour cela que l'on peut se permettre d'etiqueter un catamorphisme par la flèche de jonction de l'algèbre d'arrivée (sur l'exemple : $(\!|\ \eta \triangledown id \triangledown +\ |\!)$).

On peut illustrer à partir de cet exemple (figure 2) la structuration à 2 niveaux, où les traits pleins représentent ce qui se passe dans la catégorie des types, et ceux en pointillés, ce qui se passe dans la catégorie des $T_{\mathbf{N}}$−algèbres.

Les catamorphismes dans $T_A$-Alg représentent une bonne partie des programmes qu'il nous est possible d'exprimer à partir du type construit $A\varpi$. Par ailleurs, l'algèbre $(T_A A\varpi \rightarrow A\varpi)$ est initial dans cette catégorie (il existe une unique flèche -donc un catamorphisme- entre cet objet et tout autre objet appartenant à $T_A$-Alg). Cette initialité se justifie par le fait que $(T_A A\varpi \rightarrow A\varpi)$ est la limite du foncteur $T_A$ (cf. [BAN 93]). Ainsi, tous les programmes représentant des homomorphismes sur $A\varpi$ existent. Si on veut les découvrir, il suffit de décrire le monoïde d'arrivée. De là, on obtient même un moyen (récursif) pour exprimer ce programme. En effet, un catamorphisme $h$ peut se calculer par la composition des 3 étapes suivantes (cf. figure 2) :

$$f \cdot (id + id + h \times h) \cdot (\alpha_A^{-1}),$$

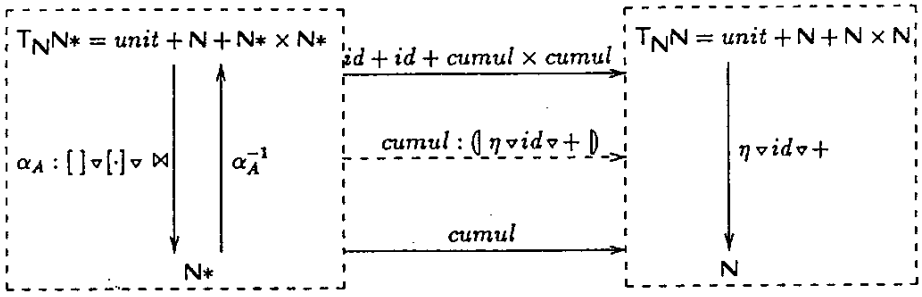



**Figure 2.** *Exemple de catamorphisme*

où $\alpha_A^{-1}$ représente la décomposition du type construit, $(id+id+h\times h)$ est soit l'application à la liste vide, soit l'application à un singleton, soit l'application récursive de $h$, et $f$ rassemble les résultats. Remarquons que ce procédé de calcul suit le schéma classique "divide-and-conquer" bien connu en parallélisme.

En fait, l'exemple présenté a la particularité d'être une réduction car la fonction $f_2$ est $A \longrightarrow A$ (ce qui n'est pas forcément vrai pour un catamorphisme quelconque). Une réduction touche à la structure du type, sans toucher au type des éléments qui composent le type. Dans ce cas, $h$ que l'on note $f/$

$$f/ = (id \triangledown f) \cdot (id + id + f/ \times f/) \cdot (\alpha_A^{-1}).$$

### 2.3. *Polymorphisme*

Rien, dans la construction que nous avons présentée, ne dépend de propriétés du type $A$. Exactement la même construction peut être effectuée à partir d'un type de base $B$, donnant une catégorie de $T_B$−algèbres. Nous pouvons définir une catégorie T-Alg dont les objets sont les algèbres initiales construites sur les différents types de base $(T_A A\varpi \longrightarrow A\varpi; T_B B\varpi \longrightarrow B\varpi; \ldots)$.

A toute fonction $f : A \longrightarrow B$ dans la catégorie des types, on associe une flèche notée $f\varpi$ dans T-Alg : $f\varpi : A\varpi \longrightarrow B\varpi$. Remarquons que $\varpi$ est un foncteur (communément appelé Map) de la catégorie des types dans T-Alg (pour les listes, une fonction $f : \mathsf{N} \longrightarrow \mathsf{R}$ par exemple, se transforme en $f* : \mathsf{N}* \longrightarrow \mathsf{R}*$). $f\varpi$ peut aussi être vue comme un catamorphisme entre les 2 algèbres, nous donnant ainsi un schéma récursif de calcul :

$$f\varpi = \alpha_B \cdot (id + f + f\varpi \times f\varpi) + \alpha_A^{-1}.$$

$f\varpi$ change le contenu d'un type structuré, sans en altérer la structure. Remarquons cependant que l'implantation parallèle d'un Map ne nécessite pas a priori de décomposer récursivement le type, mais peut être effectuée en une seule étape de calcul en appliquant $f$ à tous les éléments du type en même temps.

### 2.4. *Bilan de la construction*

Ces 2 types de calculs hautement parallèles que sont Map et Reduce permettent d'exprimer des homorphismes sur un type construit.

A ce propos, par "promotion", la composition d'un catamorphisme et d'un homomorphisme donne un nouveau catamorphisme.

Comme les diagrammes représentant les relations entre T-algèbres le matérialisent (cf. figure 2), nous obtenons des équations exprimant le calcul sur le type structuré.

De plus, si le type construit est "séparable" [SKI 93a] alors *tout* homorphisme a la particularité de pouvoir s'exprimer comme la composition d'un Map et d'un Reduce : $g/ \cdot f\varpi$, et toute fonction injective est un homomorphisme (résultats bien connus dans la théorie des listes [BIR 87]).

Soit par exemple l'opération "inits" dans la théorie des listes, qui calcule les segments initiaux d'une liste :

$$inits[a_1, a_2, \ldots, a_n] = [[a_1], [a_1, a_2], \ldots [a_1, a_2, \ldots, a_n]].$$

Inits est injective, donc donne lieu à un catamorphisme. D'après les résultats qui viennent d'être énoncés, on sait qu'il existe un autre catamorphisme, que l'on appelera "préfix" qui est

$$\oplus// = (\oplus/) * \cdot inits$$

où $\oplus/$ est une réduction quelconque. Inits lui-même peut s'exprimer comme la composition d'un Map et d'un Reduce (voir [BIR 87]).

Un point non (encore) couvert par la construction est l'interaction entre types de structures différentes.

## 3. Discussion

Etudions à présent l'adéquation de notre approche vis-à-vis des cinq propriétés énoncées.

### 3.1. *Aspect programmation*

Un programme dans notre modèle s'exprime par la composition de fonctions. Il est clair qu'un tel programme est bien indépendant des architectures puisqu'il peut même être porté sur une architecture séquentielle.

Le degré de parallélisme, éventuellement massif, et sa gestion n'apparaissent pas au niveau du programme. Ainsi, le programmeur ne se soucie ni de la décomposition en tâches et de leur placement, ni de communication et de synchronisation entre elles. Ce sera pris en compte à un niveau plus bas (voir section 3.2).

Notre but est d'aboutir à une méthode complète de développement de logiciel parallèle, centrée autour de l'approche présentée à la section 2, schématisée par la figure 3, où seuls les derniers niveaux sont influencés de façon plus ou moins directe par l'architecture cible.

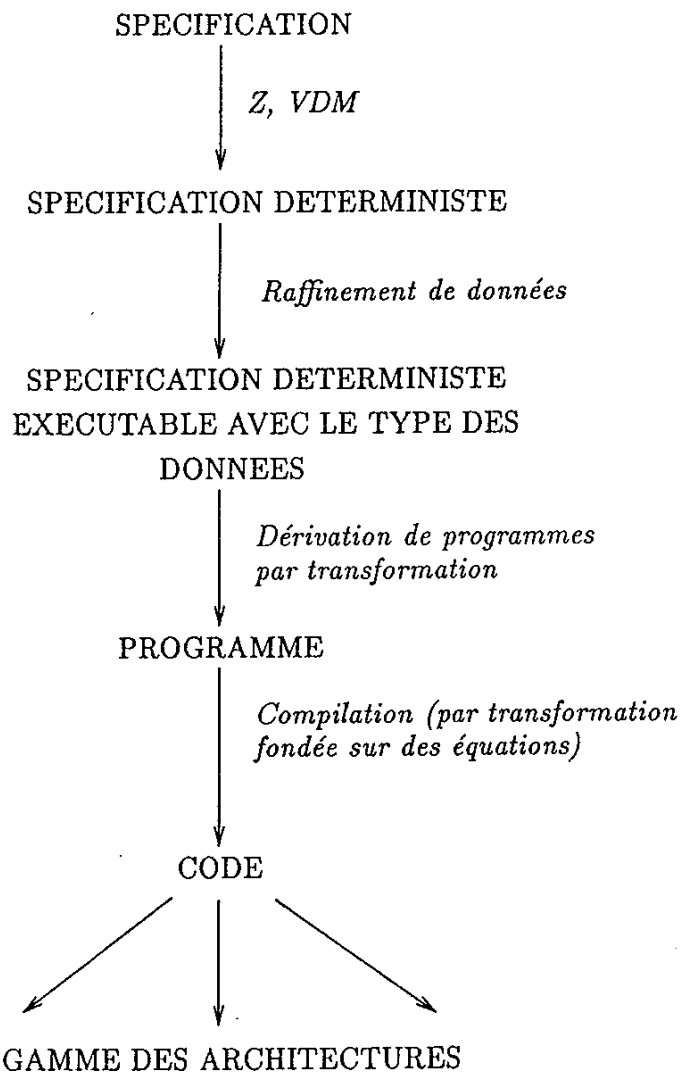



**Figure 3.** *Méthode de développement de logiciel*

Le raisonnement de haut niveau sur le logiciel peut s'effectuer grâce à des approches bien comprises maintenant, telles Z [SPI 89] ou VDM [KIN 90]. Une fois qu'une spécification a pu être raffinée jusqu'à être déterministe, des décisions concernant les algorithmes ou les structures de données peuvent être entreprises. Le domaine du raffinement des données (dans l'optique de choisir un type de données adapté au problème à résoudre) reste cependant encore peu exploré. Lorsque le type est déterminé, la construction catégorique correspondante s'applique.

L'essentiel des travaux, qui à partir d'une spécification dérivent un programme utilisant des Map et Reduce, concernent le type liste (tel celui présenté à la section 2). De tels programmes sont corrects par rapport à la spécification de départ, du fait que toutes les expressions différentes d'un même catamorphisme (celui que l'on cherche à exprimer comme la composition de Maps et de Reduces) sont reliées par des équations. L'intelligence du programmeur est mise à contribution car c'est lui qui guide la transformation, c'est-à-dire, qui définit les chemins qu'il veut emprunter afin de se rendre à un point le satisfaisant. Cela consiste à définir les fonctions de base qui font voir le jour aux différentes relations entre les différentes T-algèbres (voir par exemple [BIR 89], [BIR 87]). La notion de satisfaction fait appel à celle d'efficacité d'un calcul. Dans cette optique, chaque membre d'une équation pouvant être muni d'un coût (cf. section 3.2), on préfèrera bien sûr, celui dont le coût est moindre.

### 3.2. *Implantation du modèle - Mesures de coût*

Un calcul résulte le plus souvent d'une interaction entre les données auxquelles on l'applique. Au moment de sa conception, il est parfaitement naturel de supposer qu'une interaction entre données constitue un pas du calcul, au même titre que le travail effectué sur des données isolées (hypothèse vérifiée dans tous les modèles de calcul séquentiels, ainsi que dans tous les modèles d'algorithmique parallèle, en particulier en PRAM et comme nous le souhaitons, dans notre formalisme). Cependant, bien que cette hypothèse reste valide sur les architectures séquentielles, ce n'est pas le cas pour les machines parallèles. De plus, l'écart entre cette hypothèse et la réalité diffère selon l'architecture parallèle.

Les deux paramètres importants d'un calcul parallèle sont $vc$ le nombre de processeurs (virtuels) que l'on utilise, $sc$ le nombre de pas de calcul, le produit $vc \cdot sc$ indiquant la borne supérieure au nombre total d'opérations [BLE 90]. La tâche (parfois impossible à mener à bien) de l'implanteur est de faire en sorte que $p$ le nombre de processeurs physiques $\times$ $t$ le temps soit égal à $O(vc \cdot sc)$, sous l'hypothèse (relativement réaliste) qu'un pas de calcul sur un processeur coûte 1 et qu'un échange de donnée entre 2 entités physiques directement connectées coûte 1.

Séparons en deux classes les architectures parallèles : celles dont le réseau de communication entre $p$ processeurs comporte

— $O(p \cdot \log p)$ moyens de communication (exemple : machines à mémoire partagée à $p$ processeurs bâties autour d'un réseau de $\log p$ étages tel un butterfly, machines hypercubiques, ...);

— $O(p)$ moyens de communication (exemple : cube-connected cycle [PRE 81], shuffle-exchange, butterfly, grilles, ...).

Pour la première classe, les travaux de Valiant [VAL 90b] montrent qu'un quelconque pas de communication parallèle où au plus $O(\log p)$ messages sont issus de chacun des $\frac{p}{\log p}$ processeurs et au plus $O(\log p)$ messages sont destinés à chacun des $\frac{p}{\log p}$ processeurs, prend un temps $O(\log p)$ avec une forte probabilité. Ainsi, $((p \times t) = O(p)) = (O(vc \times sc) = p \times 1)$. Ce résultat est au cœur de l'implantation efficace de n'importe quel algorithme parallèle, y compris ceux exprimables dans notre formalisme. De ce fait, nous ne reviendrons pas sur cette classe d'architectures.

Pour la deuxième classe, Skillicorn [SKI 90] montre pourquoi un tel résultat s'appliquant au cas général ne peut exister. En effet, l'implantation d'un quelconque pas de communication peut nécessiter $O(vc \times diam)$ opérations, où $diam$ est la distance maximum séparant deux processeurs, alors que le réseau de communication sur lequel seraient branchés $p$ processeurs n'est capable de supporter que $O(p)$ messages à la fois. Ainsi $((p \times t) = O(p \times diam)) > (O(vc \times sc) = p \times 1)$.

Notre seul espoir est de restreindre le modèle de calcul à des primitives dont l'implantation par l'une des trois techniques qui vont être énoncées, soit moins gourmande en temps de communication.

1° **Plongement** La topologie des communications entre les processeurs virtuels se plonge parfaitement dans celle du réseau de communication. Un pas de communication se traduit alors par au plus un échange entre deux processeurs.

2° **Placement et ordonnancement** des processeurs virtuels de telle sorte que tout le calcul soit terminé en temps $O(sc \times vc)$.

3° **Ecriture d'un algorithme spécifique** adapté à l'architecture cible réalisant un calcul équivalent, de complexité $p \times t = O(vc \times sc)$.

Regardons à partir d'exemples ce qui se passe pour la construction que nous proposons : Les catamorphismes que sont les réductions dans le cas des collections de valeurs (listes, ensembles par exemple) où l'opérateur de réduction est tel que la taille des opérandes reste invariante, ont un graphe de communication ayant la forme d'un arbre binaire équilibré. Leur coût est $vc = n$, $sc = \log n$. Ainsi, elles s'implantent efficacement sur les architectures pour lesquelles on peut plonger un tel arbre dans la topologie physique (technique 1).

Le calcul de l'homomorphisme particulier sur les listes, qu'est le filtre

$$p \trianglelefteq [a_1, a_2, \ldots a_n]$$

a pour effet de supprimer de la liste les éléments sur lesquels le prédicat $p$ est faux. Ce calcul s'exprime par la construction catégorique comme l'application du test du prédicat sur chaque élément, suivie par la constitution d'une nouvelle liste (une liste sera implantée comme une collection de valeurs placée sur des processeurs adjacents ordonnés). La constitution d'une nouvelle liste est une opération de réduction selon l'opérateur de concaténation dans le cas des listes, mais qui accroît la taille des opérandes à chaque étape. Le calcul sous cette forme est donc de coût $vc = n$, $sc = O(n \cdot \log n)$.
Une autre implantation consiste à compter combien d'éléments subsistent après l'application du test de $p$, puis à déplacer chacun vers le processeur approprié, de telle sorte que

les éléments se retrouvent sur des processeurs voisins dans l'ordre qui leur est associé. Le graphe de communication dépendant du contenu de chaque liste n'étant connu qu'à l'exécution, la technique de plongement ne peut pas s'appliquer. Par contre, ce schéma de communication possède une solution spécifique de temps $O(\log p)$ sur de nombreuses architectures distribuées [NAS 81] (nous utilisons alors la technique 3). Ainsi, nous avons amélioré le coût de l'opération filtre dans le modèle ($vc = n$, $sc = \log n$), et proposé une implantation qui le préserve.

Dans le même ordre d'idées, une opération de tri peut être exprimée par notre construction comme étant de la forme :

$$sort = Y/ \cdot [\cdot]*$$

où Y est la fusion de 2 listes triées en une liste triée. Clairement, cette façon de calculer ne mène pas à un algorithme dans $\mathcal{NC}$ (c'est-à-dire de temps parallèle $\log^k n, k > 1$). La construction ne permet pas de définir de permutations d'une collection de valeurs d'une autre manière. Si de telles opérations se révèlent vraiment utiles pour certaines applications, on peut adjoindre au modèle certaines opérations algébriques, leurs équations et leurs coûts réalisant des opérations géométriques bien choisies en temps polylogarithmique identique quelle que soit l'architecture cible [KUM 93].

L'étude de l'implantation de catamorphismes sur les types arbre et graphe sont en cours. Un résultat récent [MEY 92] nous autorise à associer aux opérations de réduction d'arbres binaires, le coût $vc = O(n)$, $sc = O(\log n)$, reflétant ainsi les performances à l'exécution sur au moins un membre de chaque classe d'architectures parallèles.

**Calcul de coût** Nous sommes donc en mesure d'étiqueter les équations déduites automatiquement de la construction par des mesures de coût. Nous disposons pour l'instant d'un système de calcul à partir de tels coûts, appliqué aux listes, qui doit aider dans l'activité de dérivation des programmes [SKI 93b]. Lorsque nous avons une implantation de meilleur coût pour une opération particulière, il suffit de rajouter une équation reflétant ceci, dans le système de calcul des coûts. Ce système peut également être étendu en vue de la génération de code à l'aide d'équations (étiquetées elles aussi par des coûts) décrivant par exemple comment implanter une primitive sur une architecture dont le nombre de processeurs $p$ sera différent de $vc$.

## 4. Conclusion

La construction catégorique et l'environnement dans lequel nous la concevons, répondent de manière assez satisfaisante aux propriétés recherchées.

Pour ce faire, notre modèle met en relation deux mondes qui souvent s'ignorent.

Mais nous nous heurtons naturellement au dilemme que nous avons essayé d'évoquer par les exemples cités à la section 3 : la difficulté de déduire du modèle de programmation une implantation des calculs qui soit satisfaisante, de telle sorte que l'on ne soit pas tenté d'avoir un algorithme spécifique pour chaque opération. En effet, cela nuirait à un aspect important qui est la facilité de portabilité du modèle. Un compromis ne pourra être trouvé qu'au vu d'expérimentations dans divers domaines d'application.

## 5. Bibliographie


[AKL 89] S.G. AKL. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[BAN 91] J.P BANÂTRE et D. Le MÉTAYER. <Introduction to Gamma>. Dans *Research directions in High-Level Parallel Programming Languages, LNCS 574*, pages 197–202, 1991.

[BAN 92] C. BANGER. <Arrays with categorical type constructors>. Dans *Proc. of the ATABLE workshop on arrays*, 1992.

[BAN 93] C.R. BANGER et D.B. SKILLICORN. <Constructing Categorical Data Types>. Rapport de Recherche Interne Queen's University, 1993.

[BAU 91] F. BAUDE. <*Utilisation du paradigme acteur pour le calcul parallèle*>. Thèse de Doctorat, LRI, Université Paris-Sud, Orsay, 1991.

[BIR 87] R.S. BIRD. <An introduction to the theory of lists>. Dans *Logic of programming and calculi of discrete design*, pages 3–42. Springer-Verlag, 1987.

[BIR 89] R. BIRD, J. GIBBONS, et G. JONES. <Formal derivation of a pattern matching algorithm>. *Science of Computer Programming*, 12:93–104, 1989.

[BLE 90] G.E BLELLOCH. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge Massachusetts, 1990.

[CHA 88] K.M. CHANDY et J. MISRA. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[COL 89] M. COLE. *Algorithms Skeletons : Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[CUL 93] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K.E. SCHAUSER, E. SANTOS, R. SUBRAMONIAN, et T. von EICKEN. <LogP : Towards a Realistic Model of Parallel Computation>. Dans *Proceed. of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[DAR 93] J. DARLINGTON, A. FIELD, P. HARRISON, P. KELLY, D. SHARP, Q. WU, et R. WHILE. <Parallel Programming using Skeleton Functions>. Dans *Proceedings of PARLE 93*, pages 146–160. LNCS, 1993.

[GIB 91] J. GIBBONS. <*Algebras for Tree Algorithms*>. PhD thesis, Oxford Univ., Oxford, UK, 1991.

[KIN 90] S. KING. <Z and the Refinement Calculus>. Dans *VDM'90 : VDM and Z - Formal Methods in Software Development, LNCS 428*, pages 164–188. Springer Verlag, 1990.

[KUM 93] K.G. KUMAR et D. SKILLICORN. <Data parallel geometric operations on lists>. Rapport de Recherche Interne Queen's University, 1993.

[LAM 86] J. LAMBEK et P.J. SCOTT. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[MAL 90] G. MALCOLM. <*Algebraic Data Types and Program Transformation*>. PhD thesis, Rijksuniversiteit, Groningen, 1990.

[MEY 92] E. MEYR et R. WERCHNER. <Optimal Routing of Paranthesis on the Hypercube>. Dans *Proc. of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 109–117, 1992.

[NAS 81] D. NASSIMI et S. SAHNI. <Data broadcasting in SIMD computers>. *IEEE Transactions on Computers*, 30:101–106, 1981.

[PAR 92] N. PARIS. <Définition de POMPC (version 1.99)>. Rapport de Recherche No. LIENS-92-5, LIENS, Paris, 1992.

[PIE 91] B.C. PIERCE. *A Taste of Category Theory for Computer Scientists.* MIT Press, Cambridge Massachusetts, 1991.

[PRE 81] F.P PREPARATA et J. VUILLEMIN. ≪The Cube-Connected Cycles : A versatile network for parallel computation≫. *Comm. of the ACM*, 24:300–309, 1981.

[SKI 90] D.B. SKILLICORN. ≪Architecture-Independant Parallel computation≫. *IEEE Computer, December*, 23:38–50, 1990.

[SKI 93a] D.B. SKILLICORN. ≪Categorical Data Types≫. Dans *2nd workshop on abstract machine models for highly parallel computers.* Oxford University Press, 1993.

[SKI 93b] D.B. SKILLICORN et W. CAI. ≪A cost calculus for parallel functional programming≫. Rapport de Recherche ISSN-0836-0227 93-348, Department of Computing and Information Science, Queen's University, Kingston, Canada, 1993.

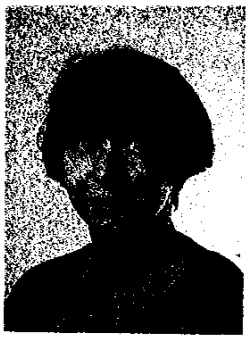
[SPI 89] J.M. SPIVEY. *Understanding Z: A Sprecification Language and its Formal Semantics.* Cambridge University Press, 1989.

[STE 86] G. STEELE et W. HILLIS. ≪Connection machine Lisp : fine-grained symbolic processing≫. Dans *Symp. on Lisp and Functional Programming*, pages 279–297, 1986.

[VAL 90a] L.G. VALIANT. ≪A bridging model for parallel computation≫. *Communications of the ACM*, 33:103–111, 1990.

[VAL 90b] L.G. VALIANT. Dans *Handbook of theoretical computer science*, volume B, Chapître ≪General purpose parallel architectures≫, pages 943–972. Elsevier, 1990.

Article reçu le 6 avril 1993, version révisé le 14 septembre 1993.



***Françoise Baude*** *est maître de conférences à l'université de Nice Sophia-Antipolis. Sa thèse en informatique de l'université de Paris XI, en 1991, est dédiée aux modèles de calcul parallèle présentant une indépendance vis-à-vis des architectures. Ses recherches sont désormais axées vers l'implantation optimisée des primitives abstraites que l'on trouve dans ces modèles.*

***David Skillicorn*** *a obtenu un B.Sc de l'université de Sydney, Australie, en 1978 et une thèse de doctorat de l'université du Manitoba, Canada, en 1981. Il est professeur associé à l'université de Queen, où ses travaux concernent le calcul parallèle d'intérêt général.*

## 2.2 Programmation à objets parallèle

### 2.2.1 Distributed objects for parallel numerical applications

F. Baude, D. Caromel, and D. Sagnol. Distributed objects for parallel numerical applications. *Mathematical Modelling and Numerical Analysis Modelisation, special issue on Programming tools for Numerical Analysis, EDP Sciences, SMAI*, 36(5) :837–861, 2002.

**Mathematical Modelling and Numerical Analysis**
Modélisation Mathématique et Analyse Numérique

ESAIM: M2AN
M2AN, Vol. 36, N° 5, 2002, pp. 837–861
DOI: 10.1051/m2an:2002039

# DISTRIBUTED OBJECTS FOR PARALLEL NUMERICAL APPLICATIONS

Francoise Baude[1], Denis Caromel[1] and David Sagnol[1]

**Abstract.** The C++// language (pronounced C++ *parallel*) was designed and implemented with the aim of importing *reusability* into parallel and concurrent programming, in the framework of a MIMD model. From a reduced set of rather simple primitives, comprehensive and versatile libraries are defined. In the absence of any syntactical extension, the C++// user writes standard C++ code. The libraries are themselves extensible by the final users, making C++// an open system. Two specific techniques to improve performances of a distributed object language such as C++// are then presented: Shared-on-Read and Overlapping of Communication and Computation. The appliance of those techniques is guided by the programmer at a very high-level of abstraction, so the additional work to yield those good performance improvements is kept to the minimum.

**Mathematics Subject Classification.** 68N15, 68N19.

Received: 11 December, 2001. Revised: 23 May, 2002.

## 1. Introduction

Reusability has been one of the major contributions of object-oriented programming; bringing it to parallel programming is one of our main goals, and a major step forward for software engineering of parallel systems. Part of the challenge is to combine the potential for extensive reuse with the high performance which is usually required of parallel and real-time systems.

Working mainly within the framework of physically distributed architectures, we are concerned with both explicit and implicit parallelism in both the problem and solution domains. Our applications include parallel data structures, computer-supported cooperative work (CSCW), and fault-tolerance and reliability in safety-critical and real-time systems.

To achieve this end, we began design and implementation of C++// early in 1994, and we are pursuing now this research in the context of the Java language, with the definition of a library called *ProActive PDC* [18]. The C++// language benefited from previous research done on the Eiffel// language [12, 15]. Important ideas and techniques from that work have reappeared in the definition of a reduced set of simple primitives that are then composed to create comprehensive and versatile libraries, which —most importantly— can then be extended by end users.

Another important characteristic of our system is the complete absence of any syntactical extension to C++. C++// users write standard C++ code, relying on specific classes to give programs a parallel semantics. These

*Keywords and phrases.* Concurrency, data-driven synchronization, dynamic binding, inheritance, object-oriented concurrent programming, polymorphism, reusability, software development method, wait-by-necessity, overlap, object sharing.

[1] OASIS, Joint Project CNRS, INRIA, University of Nice Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 Valbonne Cedex, France. e-mail: Denis.Caromel@sophia.inria.fr

programs are then passed through a pre-processor, which generates new files. The original and new code is then compiled and linked with a standard C++ compiler. When appropriate, all names related to the `C++//` system include the `ll` root in their name (for "parallel"). During the presentation of our system, we will conform to the following symbols when introducing:

a *model* principle or rule: ◊ a new *syntax*, class, or member: Ⓢ
a *file* used in our system: ⊡ *examples*: ∇

We hope these conventions ease reading and quick referencing through the paper.

This article begins by describing the basic features of our programming model, which is an MIMD model without shared memory. Section 3 deals with the control programming of processes, *i.e.* the definition of concurrent process activity. A recommended method for parallel programming in `C++//` is outlined in Section 4.1. Those parts of the programming environment which handle compilation and mapping are described in Section 4.2, and an overview of the implementation techniques which make the system open and user-extensible is given in Section 4.3. Finally, we present two performance optimizations that to be applied demand some implication from the user, but at a high-level of abstraction: on one side, sharing objects among processes in case they are in the same address space, on the other side, overlapping communication with computation in the framework of remote method calls. This paper ends up with conluding remarks in Section 7.

## 2. Basic model of concurrency

This section describes four important characteristics of our parallel programming model: parallel processes, communication between them, synchronization, and data sharing. As described below, we adopt a MIMD model without shared memory, which means that there are no directly-shared objects in our system.

Along with simplicity and expressiveness, reusability is one of our major concerns. More specifically, we want to allow users to take an existing C++ system and transform it into a distributed one, so that they may derive parallel systems from sequential ones [14].

### 2.1. Processes

One of the key features of the object-oriented paradigm is the unification of the notions of module and type to create the notion of class. When adding parallelism, another unification is to bring together the concepts of class and process, so that every process is an instance of a class, and the process's possible behavior is completely described by its class.

◊ *Model:* the process structure is a class; a process is an object executing a prescribed behavior.

However, not all objects are processes. At run-time, we distinguish two kinds of objects: *process objects* (or active objects), which are active by themselves, with their own thread of control, and *passive objects*, which are normal objects. This second category includes all non-active objects. An example of the arrangement of processes and objects at run-time is given in Figure 1.

At the language level, there are two ways to generate active objects. In the first, an active object is obtained by instantiating a standard sequential C++ class using `Process_alloc`:

Ⓢ *Syntax:*

```
A* p;          // A is a normal sequential class
p = (A *) new Process_alloc ( typeid (A), ...);
```

In this case, a standard sequential class `A` is instantiated to create an active object, which then has a FIFO synchronization: method invocations are serviced in the order in which they are made. The `Process_alloc` class is part of the `C++//` library, while `typeid` is the standard C++ run-time type identification (RTTI) operator. We will refer to this technique as the *allocation* style of process creation, and say that it produces an *allocation process*, or *allocation active object*. The allocation style is convenient, but limited because it only allows us to create processes with a FIFO behavior.

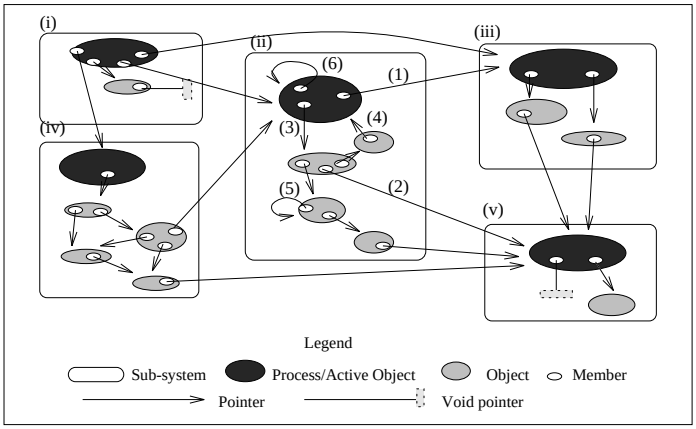



FIGURE 1. Processes and objects at run-time.

The second technique, which we call *class-based*, is more general:

◊ *Model:* all objects which are an instance of a class which publicly inherits from the `Process` class are processes.

This `Process` class is part of the `C++//` library. To use class-based process creation, the programmer must therefore derive a specific class, called a *process class*, from `Process`, as in:

Ⓢ *Syntax:*

```
class Parallel_A : public A, public Process {
  ...
};
⋮
Parallel_A* p;
p = new Parallel_A(···);
```

As with the allocation-based technique, instances of sub-classes of `Process` have a default FIFO behavior. However, as we will see in the following sections, it is possible to change this to create other behaviors. We say that the class-based technique generates *class-based processes*, or *class-based active objects*.

As shown in Figure 1, passive objects (*i.e.*, objects which are not active) belong at run-time to a single process object. This organizes a parallel program into disjoint sub-systems, each of which consists of one active object encapsulating zero or more passive objects. Figure 2 presents the two styles of active object definition.

## 2.2. **Sequential or parallel processes**

A major design decision for any concurrent programming system is whether processes are sequential (*i.e.*, single-threaded) or able to support internal concurrency (*i.e.*, multi-threaded). Because our system is oriented towards reuse and software engineering of parallel systems, rather than operating systems programming, we made the following choice:

◊ *Model:* a process is sequential, it is single-threaded.

We believe that single-threaded processes are easier to reuse, and easier to write correctly.

The model does not allow the user to program multi-threaded processes, but this does not prevent multi-threading at the operating system level. As we will see in Section 4.2, several sequential processes can be implemented with one multi-threaded operating system process for the sake of light-weightness.

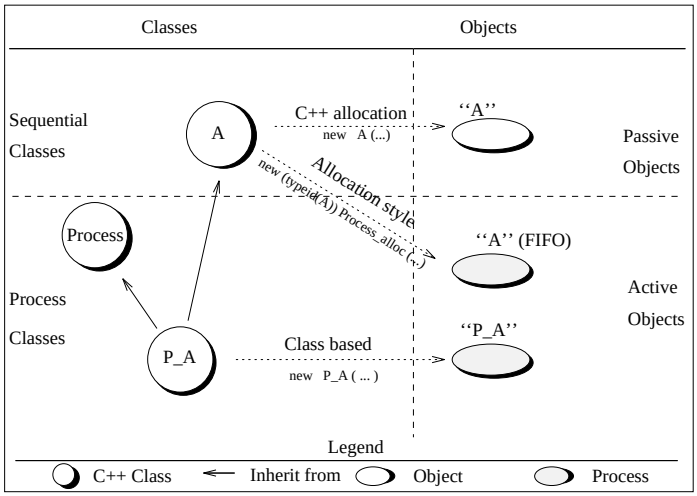



FIGURE 2. Allocation and class based active objects.

## 2.3. **Communication**

Since a process is an object, it has member functions. When an object owns a reference to a process, it is able to communicate with it by calling one of its public members. This is `C++//`'s inter-process communication (IPC) mechanism:

◊ *Model:* communications towards active objects appear syntactically programmed as member function calls.

The syntax of an IPC is unified with a standard call:

Ⓢ *Syntax:*

```
p -> f ( parameters );
```

This idea, introduced by the Actors model [3, 30], means that what is sometimes called a process entry point is identical to a normal routine or member function.

While this idea is widely used in parallel object-oriented systems, there are many differences in the definition of the semantics of method-based IPC. In `C++//`:

◊ *Model:* communications are asynchronous between processes.

Function calls towards passive objects retain the synchronous semantics of standard C++. This choice encourages parallel execution of objects, and makes each process code more independent and self-contained. As we will see further, it is also very important for supporting reusability. Synchronous function call is also possible in `C++//`, but must be specified explicitly in either the function call or the process definition.

Systematic asynchronous IPC structures a `C++//` system into independent asynchronous *sub-systems*: all the communications between sub-systems are asynchronous. Figure 1 demonstrates five sub-systems.

## 2.4. **Synchronization**

Asynchronous communication can be difficult for programmers to manage. For instance, since even function calls to processes are asynchronous, before using result values one usually needs to explicitly add synchronizations in order to make sure they have been returned by the processes. Commonly, such models lack synchronization. We use a simple rule to address this drawback: *wait-by-necessity*.

◊ *Model:* a process is automatically blocked when it attempts to use the result of a parallel member function call that has not yet been returned.

Thus, a caller does not wait for the result of an asynchronous function call until that value is explicitly used in some computation. Should a value not have been returned at that point, the caller is automatically blocked until the value becomes available. This mechanism implicitly synchronizes processes; the two primitives `Wait` and `Awaited` are provided for explicit synchronization.

Ⓢ *Syntax:*

```
v = p->f(parameters);
⋮
v->foo();                // Automatically triggers a wait
                         // if v is awaited
⋮
if (Awaited(v)){        // test the status of v
  ...
}

Wait(v);                 // explicitly triggers a wait
                         // if v is awaited
⋮
obj->g(v);               // no wait if pointer access
v2 = v;
```

**Program 1.1.** Wait-by-necessity.

Program 1.1 summarizes the semantics of wait-by-necessity. The result of a function call not yet returned is called an *awaited object*. Our semantics define that no wait is triggered by assigning a pointer to such an object to another variable, or by passing such a pointer as a parameter. A wait occurs only when the program accesses the awaited object itself (which is syntactically a pointer access to the object) or transmits (copies) the object to another process.

Wait-by-necessity is a form of future [28], and is related to concepts found in several other languages: the `Hurry` primitive of Act1 [33], the `CBox` objects of ConcurrentSmalltalk [42], and the future type message passing of ABCL/1 [43]. However, an important difference is that the mechanism presented here is systematic and automatic, which is reflected in the absence of any special syntactic construction. This has a strong impact on reusability.

In order to avoid the run-time overhead involved in the implementation of wait-by-necessity, it is possible to avoid implicit synchronization and use the explicit synchronization primitives instead. This is a tradeoff between programming ease and reusability on one hand, and efficiency and speedup on the other.

## 2.5. **Sharing**

If two processes refer to the same object, method calls to that object may overlap, which raises all of the problems usually associated with shared data. To address this issue, each non-process object in `C++//` is a private object, and is accessible to only one process:

◊ *Model:* there are no shared passive objects.

We say that a private object belongs to its process's sub-system. The programming model ensures the absence of sharing:

◊ *Model:* the semantics of communication between processes is a copy semantics for passive objects.

- A process is an active object, sequential and single-threaded.
- Communications between active objects are syntactically programmed as member function calls, and are asynchronous.
- Wait-by-necessity: a process is automatically synchronized, *i.e.* it waits, when it attempts to use the result of a member function call that has not been returned yet.
- There are no shared passive objects.
- The semantics of communication between processes is a copy semantics for passive objects.

FIGURE 3. Basic features of the `C++//` model.

All parameters are automatically transmitted by copy from one process to another. A deep copy of the object is achieved: when an object `o` is copied, all the objects referred to by pointers in `o` are deep copied as well. The implementation automatically and transparently handles the marshalling of data and pointers implied by this, as well as circular object structures:

Ⓢ *Syntax:*

```
p -> f ( parameters ); // passive objects are automatically
                       // passed by copy between processes.
```

Processes, of course, are always transmitted by reference.

Figure 1 shows how shared objects do not appear in `C++//` programs. Each passive object is accessible to exactly one active object; each of the five sub-systems in this program consists of one active object and all the passive objects it can reach. The arcs labelled (1) and (2) are always activated as asynchronous communication (IPC), while the arcs labelled (3) to (6) are activated as normal function calls. As a consequence of the absence of shared objects, synchronization between sub-systems only occurs when one sub-system waits for a result value from another process.

Prohibiting shared data has also important methodological consequences. The absence of shared objects allows either an immediate reuse (through the automatic copy), or the identification of new processes to program in order to implement the shared objects. Finally, due to the absence of interleaving, it helps ensuring correctness of such parallel applications derived from sequential ones.

As we finished the basic characteristics of the programming model, Figure 3 summarizes them.

The model has some limitations: in order to be able to use polymorphism between standard passive objects and process objects, all public functions have to be virtual, otherwise, the non-virtual function calls will not be transformed into IPC. This drawback can be alleviated with `C++` compilers providing "*all-virtual*" option; here there is a choice between paying the price of all virtual functions, and reusability. Of course, this feature requires recompiling all files involved, but is probably a small price compared to the reuse that can be obtained.

## 3. Control programming

So far, we have only examined and defined the features of `C++//` which deal with the global aspects of the programming model, such as the nature of processes and their interactions. This section describes how the control flow of processes is specified, *i.e.* how behavior, communication, and synchronization of active objects are programmed.

### 3.1. Centralized and explicit control

Control can be decentralized, that is, distributed throughout a program or, alternatively, control can be centralized, *i.e.* gathered into one place in the definition of a process, independently of the function code.

Decentralized control makes reuse of function code difficult for two reasons. First, functions designed in a sequential framework cannot be reused in a parallel one just as they are, as elements of control must be added

to them. Second, when a new process class is obtained through inheritance, the new class often needs to change the synchronization scheme used in the original class. If control is embedded in function bodies, this may not be feasible. This leads to the following choice:

◊ *Model:* processes have a centralized control.

It allows function reuse for both sequential and process classes without changes to the bodies of such classes.

Program 1.2 presents partial code for the library class `Process`. After creation and initialization, a process object executes its `Live` routine. This routine describes the sequence of actions which that process executes during its lifetime. The process terminates when the `Live` routine completes.

Ⓢ *Syntax:*

```
class Process {
 public:
  Process (···){            // process creation
    ⋮
  }

  virtual void Live(){      // process body
    ⋮  // default FIFO behavior
  }
  ⋮
};
```

**Program 1.2.** The `Process` class.

Another design decision that must be made in concurrent object oriented systems is whether process control is implicit or explicit. Control is explicit if its definition consists of an explicitly programmed thread of control. Otherwise, control is implicit, which in practice usually means that it is declarative.

Our argument is that (see [16] for a complete discussion):

1. sometimes programmers need explicit control;
2. implicit control permits reuse of synchronization;
3. no universal implicit control abstraction exists; and
4. explicit control allows us to build implicit control abstractions.

As a consequence, the basic mechanism for programming process behaviors in `C++//` is:

◊ *Model:* explicit control.

Explicit control programming consists of defining the `Live` routine of the `Process` class and its heirs (Program 1.2) using the sequential control structures of C++. All of the expressive power of C++ is available, without any limitation. For instance, the process body of a bounded buffer can be defined as in Program 1.4.

Besides explicit control, other features are needed in order to construct abstractions for concurrent programming. These features permit `C++//` to explicitly service requests. First, defining a process's thread of control often consists of defining the synchronization of its public member functions. Since such an activity requires dynamic manipulation of C++ functions, we need:

◊ *Model:* member functions as first class objects.

In practice, only some limited features, such as the ability to use routines as parameters, and system-wide valid function identifiers, are needed.

To fill that need, we provide the function `mid()` to return function identifiers. Its usage is:

Ⓢ *Syntax:*

```
member_id f;

f = mid(put);
f = mid(A::put);
f = mid(A::put, A::get);
f = mid(A::put(int, P *));
```

In order to deal with overloading, this function returns either a single identifier, or a representation of all adequate functions.

In the same way, because we need to explicitly program request servicing, we must be able to manipulate requests as objects (*i.e.*, to pass them as parameters of other functions, to assign them to variables, and so on). We require:

◊ *Model:* requests as first class objects.

In `C++//`, a particular class ( `Request` ) models the requests; every request is an instance of this class.

Finally, to be able to fully control request servicing, programmers must have

◊ *Model:* access to the list of pending requests.

This is given through the `Process` class, with a specific member named `request_list` that contains the list.

With these three facilities in place, it is possible to program the control of processes in diverse and flexible ways.

### 3.2. **Library of service routines**

Service primitives are needed to allow programmers to program control explicitly. Usually, programmers are given only a few such primitives, mainly because they are made part of the language itself as syntactical constructions (*e.g.*, the `serve` instruction of Ada). With the primitives we define, it is possible to program a complete library of service routines [13]. Some of these are shown in Program 1.3, where `f` and `g` are member identifiers obtained from the function `mid()` introduced in the previous section.

Ⓢ *Syntax:*

```
// Non-blocking services
serve_oldest();         // Serve the oldest request of all
serve_oldest(f);        // The oldest request on f
serve_oldest(f,g, ···);   // The oldest of f or g
serve_flush();          // Serve the oldest, wipe out the others
serve_flush(f);         // The oldest on f
⋮
// Timed blocking services: block for a limited time only
tm_serve_oldest(t);     // Serve the oldest, wait at most t
tm_serve_oldest(f,t);   // The oldest request on f
⋮
// Waiting primitives
wait_a_request();       // Wait until there is a request to serve
wait_a_request(f);      // Wait a request to serve on f
```

**Program 1.3.** A library of service routines.

These functions are defined in the class `Process`, and can be used when programming the `Live` routine. There is no limitation in the range of facilities that can be encapsulated in service routines. Timed services

are an example of such expressiveness; selection based on the request parameters is another. Moreover, if a programmer does not find the particular selection function he needs, he is able to program it. Thus, libraries of service routines specific to particular programmers or application domains can be defined.

Another important point concerns efficiency: concurrent policies are determined within the context of each process, based on local information rather than by using IPC, avoiding problems like polling bias [27]. This is an important advantage in distributed programming.

```
class Buffer: public Process, public List {
  ⋮
 protected:
  virtual void Live()
  {
    while (!stop){
      if (!full)
        serve_oldest ( mid(put) );
      if (!empty)
        serve_oldest ( mid(get) );
    }
  }
};
```

**Program 1.4.** An explicit bounded buffer example.

As an illustration of the use of explicit control programming, Program 1.4 presents a `C++//` implementation of a bounded buffer. This definition implements a specific policy: when the buffer is neither `full` or `empty`, the buffer alternates service on `put` and `get`. This policy is clearly not the only possible one.

This is an example of explicitly fine-tuning the synchronization of processes. While this might be very important in some contexts, we might want to program within a more abstract framework in others, ignoring the implementation details, through the definition of libraries of abstractions [16].

As we finished the control programming of processes, Figure 4 summarizes the basic features on the `C++//` model.

- Processes have a centralized and explicit control
- Member functions and requests are first class objects.
- The list of pending requests is accessible.
- A library of service routines provides for explicit control programming.
- A library of abstractions allows for implicit and declarative control.

FIGURE 4. Control programming in `C++//`.

## 4. PROGRAMMATION, ENVIRONMENT AND IMPLEMENTATION

### 4.1. **A programming method**

Because it is rather difficult to evaluate performances of distributed system before they actually run, we believe that definition of processes has to be postponed as much as possible, and should be flexible and adaptable. The programming guide we develop in this section applies this principle, made possible by the features of the `C++//` model.

The first step is a standard, sequential, object-oriented design, possibly including object identification, interface and topology design, and sequential implementation [9, 35]. The next steps deal with parallel design and are specific to the language and technique we developed.

Processes are a subset of classes. Because object-oriented design usually gives a finer-grained decomposition than structured design or information hiding methods (because every type is a module, and every module is a type), there is no need for re-structuring. The classes remaining passive are commonly used without any changes.

Here is an example of how an entity `a` declared:

```
A* a;
```

get assigned with a process object of type `P_A` (heir of `A`):

```
a = new P_A ( ··· );
```

A function call on `a` is now executed on an asynchronous basis (the caller does not wait for its completion). This automatic transformation of synchronous call into asynchronous one is crucial to avoid routine redefinition. An inherited routine may use the result of a function call issued to the process:

```
res = a->fct( parameters );
⋮
res->g( parameters);
```

In this case, the *wait-by-necessity* handles the situation. Without this automatic *data-driven synchronization* one would have to redefine the current routine in order to add explicit synchronization.

These model properties ensure that most of the inherited routines remain valid for the process class. However, some special cases need re-programming.

Figure 5 presents the significant steps of the method.

1. Sequential design and programming.
2. Process identification.
3. Process programming:
   - Define each process class (`Process` or abstraction class).
   - Define the activity (`Live`).
   - Use the process classes with polymorphism.
4. Adaptation to constraints:
   - Refine the topology.
   - Define new Processes.

FIGURE 5. The 4 steps of the method.

## 4.2. **Environment**

This section briefly describes the facilities supporting the development of `C++//` programs, including the compilation of source code, executable generation (see Fig. 6), and more specifically a mechanism for mapping active objects onto machines.

Mapping assigns each active object created during the execution of a `C++//` program to an operating system process on an actual machine or processor. In order to avoid confusion, we call the sub-system consisting of one active and all its passive objects a *language process*, and use the term *OS process* for the usual notion of an operating system process.

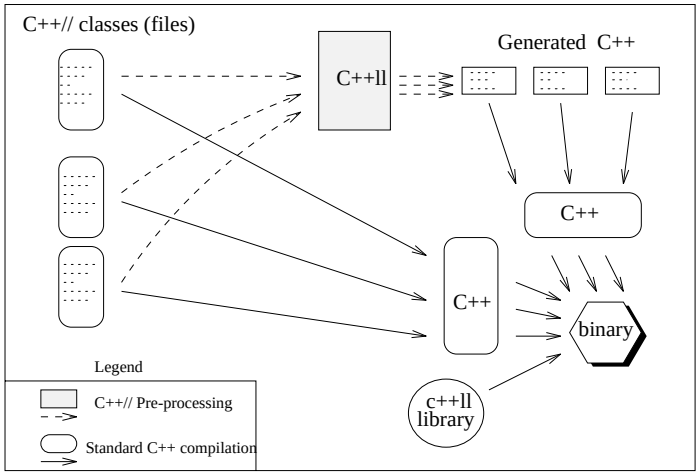



FIGURE 6. Compilation of a `C++//` system.

The mapping of a language process to an OS process on a particular processor is controlled by the programmer through the association of two criteria:

◊ *Model:*
(1) the machine where the language process is to be created;
(2) the light-weight or heavy-weight nature of the language process.

The machine itself can be specified in two ways. The first method is to specify a virtual machine name, which is simply a string. This name is related to an actual machine name by a translation file called `.c++ll-mapping`. The `C++//` system looks for this file first in the directory in which the process is running, and, if it is not found there, in the user's home directory. An example of such file is:

⊡ *File:*

```
FILE .c++ll-mapping

// virtual name   actual name
   Server         Inria.Sophia.fr
   S1             wilpena.unice.fr
   S2             192.134.39.96
   S3             // current machine
   P1             I3S-1
   ⋮              ⋮
   P6             INRIA-1
```

The other technique used to specify a machine is to use a language process that already exists. In this case, the new process is created on the machine where that language process is running. With this technique, processes can be linked together to ensure locality.

The *light-weight* switch permits creation of several language processes inside a single OS process. In the *heavy-weight* case, only one language process is mapped to each OS process. The user accesses these switches

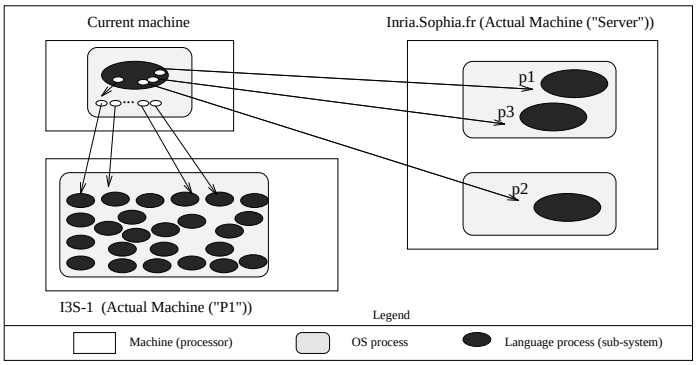



FIGURE 7. Example of mapping.

through a class called `Mapping`:

Ⓢ *Syntax:*

```
class Mapping {
 public:
  virtual void on_machine(const String& m); // set a virtual machine name
  virtual void with_process(Process* p);    // set the machine to be the same as
                                            // for the already existing process p
  virtual void set_light();                 // set to light-weight process
  virtual void set_heavy();                 // set to heavy-weight process
};
```

When a program creates a language process, an object of type `Mapping` can be passed to `new` in order to specify the desired mapping of the new process. Program 1.5 presents the syntax used for this. With the `.c++ll-mapping` file taken from above, Program 1.5 produces part of the mapping presented in Figure 7.

∇ *Example:*

```
A *p1;                // A is a normal sequential class: instantiation style
P_A *p2, *p3;         // P_A is a process class: class based style
Mapping *map1, *map2; // mapping objects
⋮
map1->set_heavy();
map1->on_machine("Server");
p1 = (A *) new (typeid(A), map1) Process_alloc(···);
                      // p1 on a new OS process,
                      // on machine with actual name "Server"
map2->set_heavy();
map2->with_process(p1); // p2 on a new heavy-weight process,
p2 = new (map2)  P_A(···); // same machine as p1
···
map2->set_light();
p3 = new (map2) P_A(···); // p3 on a light-weight process,
                        // same machine as p1,
                       // inside the same OS process as p1
```

**Program 1.5.** Mapping processes to machines.

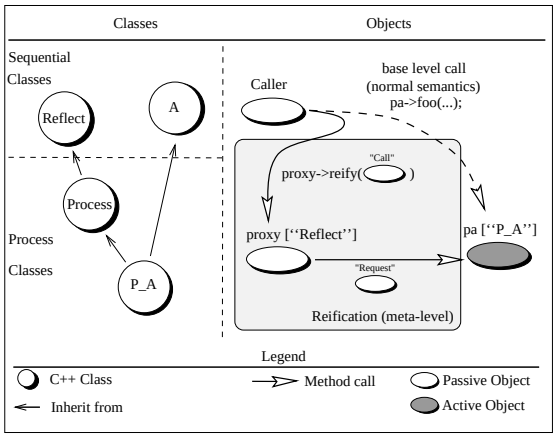



FIGURE 8. Reification of calls.

## 4.3. **Implementation**

This presentation goes beyond implementation details since the technique we use—reification—also provides for customization and extension of our system.

### 4.3.1. *A reflection-based system*

The `C++//` system is based on a Meta-Object Protocol (MOP). There are various MOPs [32], for different languages and systems, with various goals, compilation and run-time costs, and various levels of expressiveness. MOP techniques have been used in many contexts in particular for parallel and distributed programming [11, 20, 34].

Within our context, we use a reflection mechanism based on reification. Reification is simply the action of transforming a call issued to an object into an object itself; we say that the call is "reified". From this transformation, the call can be manipulated as a first class entity, *i.e.* stored in a data structure, passed as parameter, sent to another process, etc.

A *meta-object* (Fig. 8) captures each call directed towards a normal *base-level* object; a meta-object is an instance of a *meta-class*. In some ways, a *proxy*, a local object that permits to access a remote one [8, 22, 39], is a kind of meta-object.

### 4.3.2. *A* MOP *for* `C++`*: basic classes*

The main principle of our MOP for C++ is embodied in a special class, called `Reflect`, which presents the following behavior:

◊ *Model: all* classes inheriting publicly from `Reflect`, either directly or indirectly, are called *reified classes*, a reified class has *reified instances*; all calls issued to a reified object are reified.

This last requirement is important for reusability, as it permits users to take a normal class, and then globally modify its behavior, to transform it into a process.

Figure 8 illustrates reification. The creation of an instance of a `Reflect` class returns a meta-object (a proxy) for the type being passed in as the allocator's first parameter. From this mechanism, we implement the basic classes of our programming model described in Section 2.1.

### 4.3.3. *Customization and extension of* `C++//`

The MOP we just presented is independent of any parallel programming model. The classes of the MIMD model (such as `Process`) we described in this paper are programmed on top of the MOP, without any compiler

modification. An important consequence of this is that other parallel programming models, such as shared-memory MIMD or SPMD, can be defined on top of the MOP. The wait-by-necessity implementation, for instance, is achieved through a class `Future` which uses reification by inheriting from `Reflect`. Such an open system, or open implementation [32], is extensible by the end-user or by developers of new libraries, and adaptable to various needs and situations, such as the ones presented in the two following sections.

Notice that this MOP has been adopted as the level 0 of a standard framework for parallel C++ systems, designed in the context of the EUROPA Parallel C++ working group, formerly funded by the EU [17]. In this framework, other C++ libraries for parallel computing, such as UC++ (featuring constructs which are variations of those found in `C++//`, like active objects, asynchronous communications and futures) were implemented on top of this MOP. A similar effort in the United States is HPC++, and now, OpenHPC++ [24], which supports constructs to develop both task as well as data parallel applications in C++. [26] explains how it would be possible to implement the parallel STL model of HPC++ on top of the MOP adopted within the EUROPA working group.

They are some other few extensions of C++ for parallel computing [41] that are built upon reification mechanisms, in particular, onto the meta-object protocols OpenC++ [20] and MPC++ [31].

## 5. Sharing passive objects among active objects

This section presents a mechanism for sharing objects [19] when two active objects that reside in the same address space (in the same process) want to access the same passive object in read mode.

### 5.1. The SharedOnRead framework

A crucial point of the standard `C++//` model, is where active objects are created. A `C++//` programmer has several choices to determine the machine (or node) where a new active object will be created: (1) give the machine name, (2) provide an existing active object in order to use the same machine. But central to the issue is that, in both cases, the programmer has two options to create the new active object: (a) in an existing address space, (b) in a new address space. In case (a), several active objects will be able to share the same address space — threads belonging to a same heavy-weight process are used to implement active objects. Let us note that even when active objects in the same address space communicate, passive objects are still transmitted by copy. This potentially time and space consuming strategy is mandatory if we want the program semantics to be constant, whatever the mapping is. However, in some cases, sharing is actually possible, and copying large objects could be avoided. The SharedOnRead mechanism was defined to make possible such optimization, and to provide a general strategy that keeps the semantics unchanged when the mapping varies.

#### 5.1.1. *Strategy*

Upon a communication between two subsystems that are within the same address space, instead of copying a parameter if it is of type SharedOnRead, we just share it; otherwise, there is no alteration of the copy semantics if the two subsystems are not in the same address space (*cf.* Fig. 9). The SharedOnRead is dearly related to the *copy-on-write* mechanism that can be found in operating system (Mach [37] or Orca [6, 29] are using it). However, the strategy is slightly different in copy-on-write techniques: one wants to copy only when it is necessary, instead, with the SharedOnRead mechanism, one wants to share data on read operations whenever it is possible (*i.e.* same address space).

While this idea is quite simple, a mechanism is needed in order to maintain the copy semantics of `C++//` that should apply everywhere, even when the two subsystems are mapped in the same address space. We can notice that objects will be able to be shared by several subsystems only as long as they are not modified by one of the subsystems. In order to be accurate, the strategy needs to make a distinction between *read* and *write* accesses, and also needs to know when a subsystem *forgets* a SharedOnRead object (the subsystem suppresses its reference to this object).

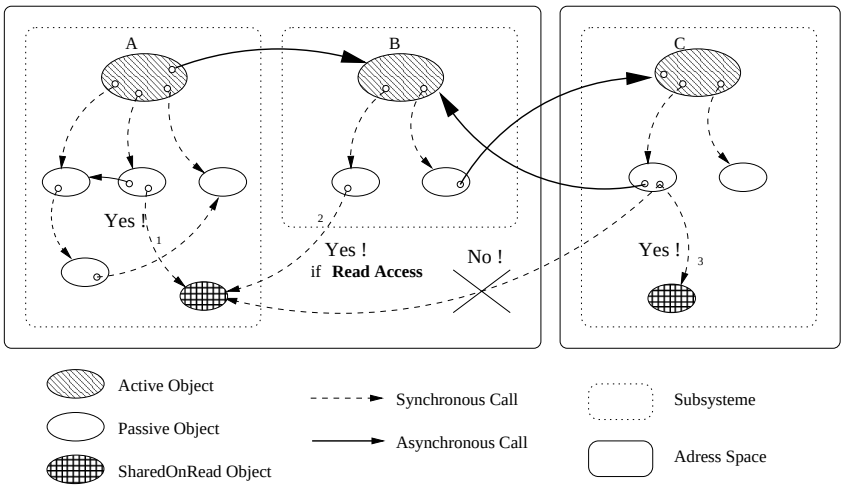



FIGURE 9. SharedOnRead objects within several address spaces.

The requirements at the design level are the following:

◊ *Model:*
(1) When a SharedOnRead object is used as a communication parameter between two subsystems being in the same address space, the original object is not copied, but instead a new reference is memorized (a counter is incremented within that object).
(2) A *read* access is freely achieved (from both the owner's subsystem or another one).
(3) Upon a *write* access (from both the owner's subsystem or another one), if the counter value is more than 1, a copy of the object is made. The modification applies to the copy. The counter of the previously existing object is decremented; the counter of the copy is set to 1.
(4) Upon a *forget* operation, the counter is just decremented. When reaching zero, the object is automatically garbaged.

5.1.2. *Programmer interface and implementation*

As a design decision, we choose to give users the control over which objects should be SharedOnRead and which should have the standard systematic copy behavior.

◊ *Model:* A SharedOnRead object is an instance of a class that inherits directly or indirectly from the `C++//` SharedOnRead class.

The SharedOnRead class is:

Ⓢ *Syntax:*

```
class SharedOnRead : public Reflect {
public:
    virtual void read_access(mid_type);
    virtual void write_access(mid_type);
    virtual void access_declaration();
    virtual void forget();
};
```

and as such provides several member functions whose usage is now described. `read_access` and `write_access`, are both used to specify how the data members are accessed by a given method. If `read_acess` is selected by

the programmer for a given method, the programmer declares that, for this function, data members are never changed, alas if `write_access` is selected, data members can be changed. These two functions take a `mid_type` parameter which is unique for all the member functions in the program; the `mid` function (see Sect. 3.2) provides that unique identifier for a function name. The `access_declaration` function has to be redefined in order to specify for each public member function its read or write nature: write access is the default behavior. A SharedOnRead user has to take care of that and it is his responsibility to check for each method in the class which ones are leaving the object in the same state and which ones are making modification to the object state. Lastly, `forget` must be called so as to declare that this SharedOnRead object is not used anymore. This have to be dealt with explicitly because SharedOnRead objects cannot be aware that they are not referenced anymore.

Program 1.6 gives an example of the use of the SharedOnRead mechanism, in which the programmer has just to define a new class that inherits from an existing class and the SharedOnRead one, the only additional programming work being to redefine the `access_declaration` function.

$\nabla$ *Example:*

```
class Block { // A Matrix block
public:
    virtual void reach(int** ai, int** aj, double** a);
    virtual void update(int**& ai, int**& aj, double**& a);
};

class SorBlock: public SharedOnRead, public Block {
public:
    SorBlock();
    virtual void access_declaration() {
       read_access(mid(reach));
       write_access(mid(update));
    }
};
```

**Program 1.6.** Programming a SharedOnRead block in a matrix-based example.

The implementation of the SharedOnRead class is based on the reification mechanism provided by `C++//`. Inheriting from the class `Reflect`, all the SharedOnRead member functions are reified: the functions are not directly executed but are derouted in a specific *proxy* where all the necessary implementation is defined and achieved (update of the counter, copy when necessary, etc.). After this step, the *proxy* executes the function.

## 5.2. **Benchmark application**

### 5.2.1. *Parallel linear algebra*

We have tested the SharedOnRead mechanism on basic linear algebra operations commonly used in iterative methods [38]. The key point for efficient parallel implementation of iterative methods are good performance of the distributed sparse matrix/vector product, and distributed dot product since these operations are at the heart of all basic Krylov algorithms [36]. In this context it is crucial to avoid unnecessary copies in matrix operations.

As defined in [36] we may focus on a reduced set of operations:

- dense SAXPY, $Y := \alpha X + \beta Y$ with $Y, X \in \mathbb{R}^{n \times p}, \alpha, \beta \in \mathbb{R}$;
- dense or sparse matrix product $Y := \alpha A \cdot X + \beta Y$ with $Y \in \mathbb{R}^{m \times p}, A \in \mathbb{R}^{m \times n}, X \in \mathbb{R}^{n \times p}, \alpha, \beta \in \mathbb{R}$.

Parallel numerical linear algebra is concerned with data distribution of the matrix arguments in the above operations. In our case we will only consider a block distribution scheme of matrices which is widely used [21] and well suited for these applications. With these kind of distributions, we have to do dense matrix saxpy operations and dense or sparse matrix products.

These operations are implemented as a method of a class `Matrix` representing a block partitioned matrix. The methods are:

- `Matrix::scal(double alpha, Matrix* B, double beta)`
  This method performs the matrix saxpy operation $this = \beta \cdot this + \alpha \cdot B$.
- `Matrix::axpy(double alpha,Matrix* A,Matrix* X,int mm)`
  This method performs the matrix operation $this = \alpha \cdot A \times X + this$, and $this = \alpha \cdot A \times X$ if `mm==1`.

Here, the distributed objects are the blocks of the matrix which may be called CSC since they are Compressed Sparse Column (potentially sparse) matrices.

5.2.2. *From sequential to parallel matrices in C++//*

A sequential matrix contains a list of CSC objects, each of these objects holding a `Block`. This `Block` is responsible for allocating all the arrays representing the matrix. If we want to parallelize these classes using C++//, we only have to redefine the `Matrix` constructors. These constructors create the distributed CSC objects of type `Block` or `SorBlock` (see Program 1.6) depending if we want to use the SharedOnRead mechanism or not. A distributed CSC object (`CSC_ll` object) can be created just by inheriting from CSC and the `C++//` class `Process`. All the functions presented above (scal, axpy, ...) come unchanged from the sequential classes.

Program 1.7 presents the sequential version for a dense axpy function, which will be reused unchanged in the `C++//` version. Thanks to polymorphism compatibility, the two A and X variables can be `CSC_ll` objects. If the SharedOnRead is used in the matrices construction, the `block()` function returns a `SorBlock` object: as such, as `bl1` and `bl2` must be accessed only in read mode, we will use them directly if they are located in the same address space, without generating any copy. On the contrary, the `mine` variable, which represents the local `Block` of the `CSC` object, will be modified, so `update` has been declared in write mode in Program 1.6.

```
void CSC::axpy(double alpha, CSC* A, CSC* X, double beta) {
   Block* bl1 = A->block();
   Block* bl2 = X->block();
   Block* mine = block();
   bl1->reach(&tia, &tja, &ta);
   bl2->reach(&tix, &tjx, &tx);
   mine->update(&tiy,&tjy,&ty);
   ...
}
```

**Program 1.7.** Sequential dense CSC_Block product.

5.2.3. *The MPI version*

Our objective is to apply the SharedOnRead mechanism and prove that this yield to performances as good as the ones obtained with MPI, but with more transparency and flexibity for the programmer due to the object-oriented model of programming. The MPI [2] implementation requires to redefine all the functions in order to take into account the fact that not the same operations must be executed depending on the processor they are executing on: MPI is very intrusive. This means that the programmer has to add a lot of MPI calls in the original sequential code in order to derive the parallel version. Moreover, this makes it difficult to the programmer to go back and forth from the sequential to the parallel version. Furthermore, MPI is a message passing library thus parallelism is explicit, and the programmer has to directly deal with distribution and communication.

5.2.4. *Performances*

The following tests were performed on a network of 4 Solaris Ultra 1 with 128MB of memory and a 10Mb/s Ethernet link. The MPI tests use the LAM library (`http://www.mpi.nd.edu/lam/`).

Since the runtime is not based on a virtual shared memory, the standard distribution algorithm can imply that several active objects (representing CSCs holding matrix blocks) get mapped within the same address space on each workstation, while others get mapped within different address spaces. Recall that the SharedOnRead

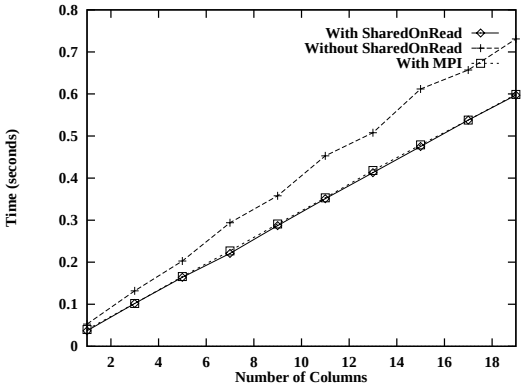



FIGURE 10. SAXPY with 4 computers.

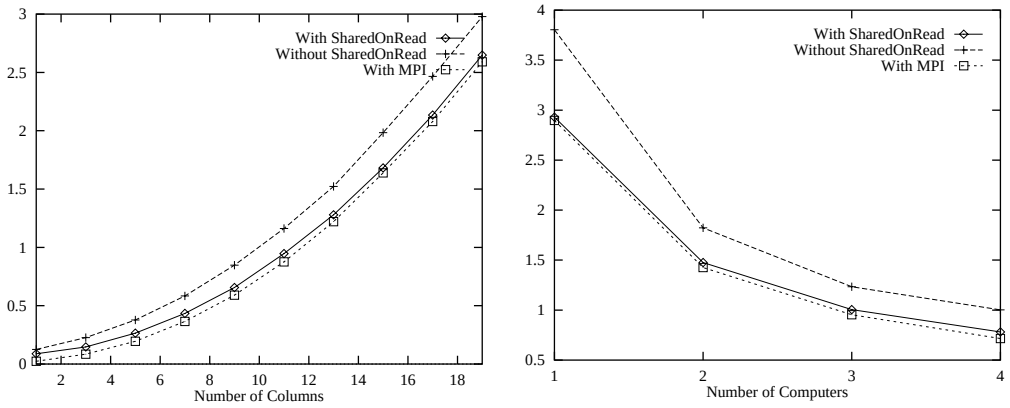



FIGURE 11. Dense matrix product (a) duration in seconds using 4 computers (b) speed-up.

optimization applies when a computation occurs within the same address space. As demonstrated below this is sufficient to achieve consequent speedup. If we were on an SMP architecture, then the benefits would be even greater since there would be opportunity for sharing all the matrix blocks.

Figure 10 presents the performances for a `scal` (*cf.* Sect 5.2.1) calculation. Matrices used during these tests were rectangular matrices with 90449 rows and a variable number of columns. The use of SharedOnRead objects demonstrates a speed-up between 20 and 25% compared to the non optimized `C++//` version. When compared with the MPI version, we cannot distinguish any difference between `C++//` with SharedOnRead and MPI. One important point to notice is the fact that the non optimized `C++//` version (without SharedOnRead objects) presents more and more overhead when the matrix size increases. The main reason is that this version requires many communications between the different active objects even if they are located in the same address space. In the SharedOnRead version, the two blocks represented by two active objects are mapped within the same process, so communications and copies are avoided.

Figure 11a presents performance results for a dense matrix product. Again, the performances of the experiment using the non optimized `C++//` version is between 20 and 30% slower than the MPI one. Between the

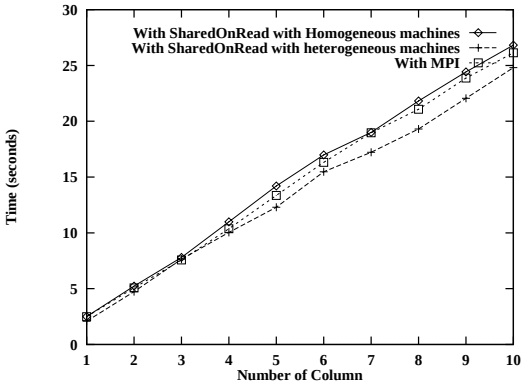



FIGURE 12. Sparse matrix product with 4 computers.

C++// version and the MPI one, the overhead is constant. But the SharedOnRead version and the MPI one do not behave exactly the same: the C++// solution sums the local matrix in a sequential way because it reuses the sequential code, whereas the MPI version requires communication during the reduction step. Figure 11b presents the speed-up obtained with 4 computers for the 3 different experiments. All calculations on dense matrices are perfectly scalable; with 4 computers, the speed-up is around 3.9. We can observe that the overhead of the non optimized C++// version is constant whatever the number of computers we use.

At last, Figure 12 presents performance results for a sparse matrix product. A first point to notice in such a case is that the *add* function of the matrix had to be rewritten in the C++// version: the reduction being critical in this benchmark, it was important to compute it in parallel. The second important aspect of this benchmark deals with the platform architecture. All the previous tests were made on homogeneous computers: the same CPU, at the same frequency, with the same amount of memory. This last test was performed on an heterogeneous platform. Two of the four Ultra 1 were replaced with two Ultra 2 with 192 MB of memory. Within this new architecture, the MPI program has the same performance as in the homogeneous test. In the C++// case, the benchmark demonstrates that the object-oriented version is actually more efficient than the MPI one. While MPI is subject to synchronization barriers, the C++// version automatically takes advantage of the asynchronous model. In that case, the reduction of the two local matrices of the fastest computers can start even if the computation on the slowest computers has not finished.

## 6. Overlapping communication with computation

This section presents the concepts and an implementation of an overlapping mechanism between communication and computation [7]. This mechanism allows to decrease the execution time of a remote method invocation, especially in the context of important transfers, such as matrices.

A general idea to lower communication costs is to overlap communication with computation, thus yielding to a pipeline effect regarding messages transmission. Any attempt to exploit this opportunity needs to rely on non-blocking elementary communications, such as for instance, asynchronous send and receive primitives as provided by well-known message-passing libraries (*e.g.* PVM [1] or MPI [2]).

For code readability and portability purposes, one additional requirement is to make the use of the overlapping technique as much transparent as possible for programmers. As such, we reject distributed hand programmed solutions where the programmer would himself split the data to be sent into smaller pieces, asynchronously send each piece in turn thus "feeding" the pipeline, while at the receiver side, explicitly and repetitively receive each new piece and goes on with it in the related computation.

Previous attempts to automatically make use of an overlapping mechanism between communication and computation have been successful in the context of data-parallel compiled languages for parallel architectures with distributed memory: HPF [10], FortranD [40], but also in LOCCS [23], a library for communication routines and computation. Here, we explain how the same problem has been tackled with, in the area of distributed object-oriented languages where the whole computation taking place on the distributed entities can be expressed as remote service invocations through method calls.

### 6.1. How the overlapping technique is designed and implemented

In the implementation of such remote method invocation-based settings, all arguments of the method call must generally be received before the method execution starts. The essence of our proposition is thus to:

◊ *Model:* apply a classical pipelining idea to the arguments of a remote call.

Once the first part of the arguments has arrived, the method execution will be able to start. Moreover, it is only the type of the arguments that will automatically indicate how to split the data to send. In this way, programmers will be able to express, at a very high level, opportunities to introduce an overlapping of communications with computation operations.

◊ *Model:* A new class, inheriting from `Reflect` (see Sect. 4.3.2) and called `later` is introduced in `C++//`, from which all objects that require to be sent latter have to inherit from.

Ⓢ *Syntax:*

```
class Matrix_Later : public later, public Matrix {...};
```

Objects from this `later` class must not be sent (eventually also, not be marshalled) during the first inspection of the objects belonging to the request, but later, each one in a new message (as would be done for $m2$ when calling `dom→ rang(m1,m2)` in Program 1.8 for example). `later` objects behave the same as `future` objects: automatic blocking when one tries to access to the value, transparent update of the object with the incoming value.

∇ *Example:*

```
class OpMatrix : public Process
  virtual int rang(Matrix *m1,
                   Matrix *m2)
    m1->square();
    m2->plus(m1);
    int res = m2->result();
    return (res);
  }
};
```

```
OpMatrix *dom =
    new ("host") OpMatrix(...);
Matrix *m1 =
    new Matrix(COLUMN, LINE);
Matrix *m2 =
    new Matrix_Later(COLUMN, LINE);
// set the values for m1 and m2
CLOCK_Call_Time_START;
int res = dom->rang(m1, m2);
CLOCK_Call_Time_STOP;
```

**Program 1.8.** Definition and use of a `C++//` remote service with a `later` parameter.

This technique applies whether objects of `later` type sit at the first level (*i.e.* they are parameters of the remote call as $m2$ in Program 1.8), or at lower levels (*i.e.* they are parts of non-`later` parameters; for example each line of a matrix could be declared `later` whereas the matrix itself not). Notice that if needed, it is possible to cast an object declared as inheriting from `later` to the original type (*e.g.* from `Matrix_Later` to `Matrix`), and vice-versa. For example, if a `later` object must be used at the very beginning of the next remote call, it would be worth to cast it now to its original type in order to send it immediately. So, to take advantage of the mechanism, the remote computation does not necessarily need a specific design (or redesign). The only

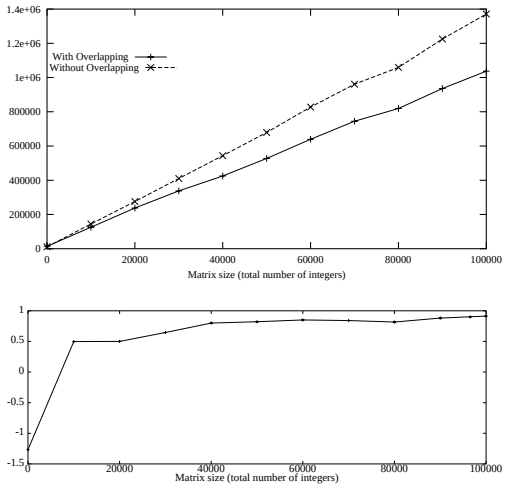



FIGURE 13. Execution of the remote service (caller side, total_duration in $\mu$s) and corresponding benefit ($G$) obtained from using the overlapping technique on a LAN.

important point is that the order the various parameters are first used should closely follow the order they are sent and received. So, the position of `later` parameters in method signatures becomes important.

Implementing the overlapping technique requires only minor modifications in the language runtime support. At the MOP level, the main modification is to write a new generic function to flatten requests: this function builds a first fragment which holds the request header and the non-`later` parameters, and then one fragment for each parameter of `later` type. Then at the runtime level, the first fragment is sent and its service will consist to create a `C++// future` type for each missing part of the request, *i.e.*, for each `later` part of the request parameters. Concerning the remaining fragments, they will be subsequently sent and served as follows: transparently update the corresponding awaited request parameters, *i.e.* the corresponding `future` objects.

## 6.2. **Benchmark**

### 6.2.1. *Description of the experiment*

We designed a simple test and benchmarked it. This test must not be considered as a real application, but as a means to validate the effectiveness of the technique. It is based on the remote call of the method `OpMatrix::rang(...)` (see Program 1.8) which takes two matrices, squares the first one, and adds the second one. As the second matrix `m2` is of type `Matrix_Later`, it can be used as a parameter of `OpMatrix::rang(...)`. The remote service can start as soon as the request id and the non-`later` parameters have been received. Experiments not using the overlapping technique are easily conducted: define `m2` as an instance of `Matrix` instead of `Matrix_Later`.

The technique should allow to overlap the transmission and reception of the `later` parameter (*i.e.* the matrix `m2`) that is only useful for the second part of the service execution (*i.e.* `m2→plus(m1)`) with the remote execution requiring only `m1` (*i.e.* the method `m1→square()`). Compared with an execution not using the overlapping technique, the duration of `m1→square()` (noted $d1$ in the following) should increase, since, at the same time, the remote processor has also to manage the reception and update of the matrix `m2`.

### 6.2.2. *Results*

Two Sun Solaris 2.6 workstations with 128 MB of RAM, interconnected by a 10 Mb/s Ethernet are used. The curves plotted in Figure 13 show a decrease of the `dom→ rang(m1,m2)` execution time, and an almost optimal gain as computed by $G$.

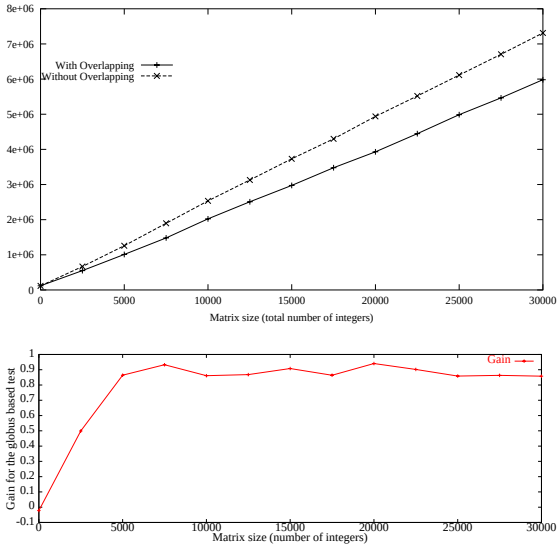



FIGURE 14. Execution of the remote service (caller side, total_duration in $\mu$s) and corresponding gain. This corresponds to one Globus-based test between USA and France during night period, with (d1) 300 times longer than in Figure 13.

Let us define the gain ($G$) in order to give a concrete estimation of the benefit.

$$G = \frac{duration_{not\_using\_overlap} - duration_{using\_overlap}}{later\_parameters\_transfer\_duration} . \quad (1)$$

The duration for transferring `later` parameters, *i.e.* `m2`, is estimated by sending a `C++//` object of the same size, not counting the—small—additional cost that would be required for managing a `later` parameter (a few milliseconds).

*Scalability.* The overlapping technique used in this context where lightweight processes are available, scales very well. Moreover, we deduce against our past experiences that only runtime supports using lightweight processes can scale so well. Indeed, benchmarks conducted in the context of `C++//` on top of PVM proved that the amount of data that could be sent and received while the remote service is in progress, is bounded by the remote receiving buffer size. The fundamental reason is that the transport-level layer can not gain the receiver process attention while this latter is engaged in a remote computation (*i.e.* `m1→square()`), due to the lack of a dedicated concurrent receiving thread.

*WAN-based results.* On WAN-based environments (see Fig. 14), sparing the transmission time of even a few bytes[1] yields a gain that the overhead of the technique can not override (very small compared to the high transmission delays). But, one should notice that the duration of the remote computation is of course an other crucial point. Indeed, if it is really too short compared with the transmission speed, almost no communication overlap occurs. This is why the Globus-based [25] grid experiment plotted in Figure 14 assigned $d1$ to be 300 times higher than in experiments plotted in Figure 13. In concrete situations, such a high-computation duration is not an unrealistic experimental assumption, as transmitting a large or even huge volume of data to remote computers (especially on a grid) is justified by the need to execute quite costly computations on these data.

[1]More precisely, the total duration for the test in Figure 14 using matrices $m1$ and $m2$ of 2500 integers decreases from 680 816 $\mu$s not using the overlapping technique to 556 446 $\mu$s when using it.

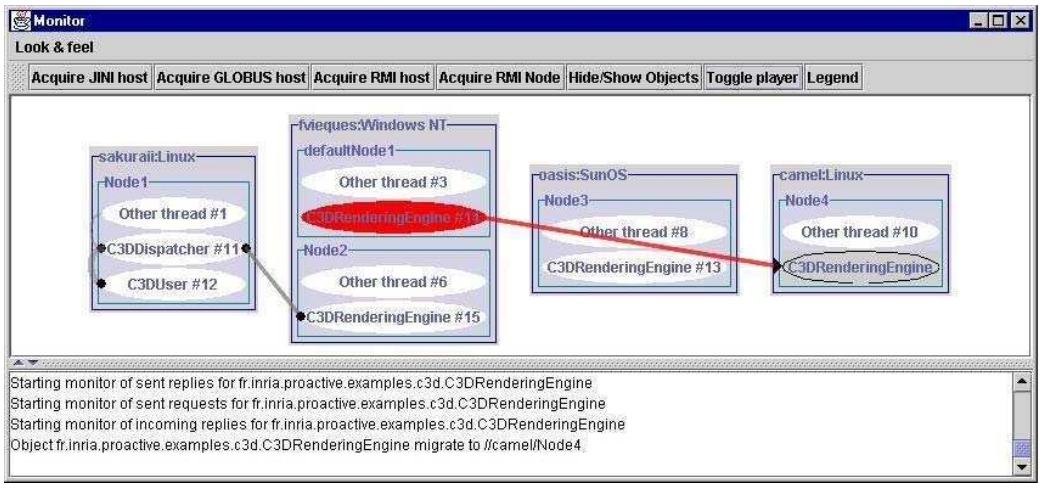



FIGURE 15. *Drag-and-drop migration* allows to graphically move objects between machines.

## 7. Conclusion and perspectives

The work presented here focussed on reuse, flexibility, and extendability. At different levels (service routines, abstractions for control programming, libraries defining specific programming models, etc.), the system we propose tries to be both abstract with information hiding principles (black boxes simple to use), and open for customization and extension. This approach tries to give some answers to the complexity and diversity of parallel programming.

Granularity is probably another crucial point of parallel programming. In order to reach performances, a challenging task is to achieve an appropriate matching between the granularity of program activities, and the capability of the underlying parallel architecture. We believe the reusability object-oriented languages make possible to be an important answer to the problem. In particuler, `C++//` makes it easy to turn an object into an active object or vice-versa, in order to adapt the granularity of program activities without not too much changes in the program.

In order to reach performances, two mechanisms have been described that can help to optimize parallel programming without too much a burden for the programmer: (1) the SharedOnRead that can help a distributed object-oriented language to be competitive with MPI, without writing large amount of code to obtain a parallel version; (2) a mechanism to overlap computations with communications in order to take advantage of pipelining in distributed object-oriented applications, without having to explicitly program the slicing of data and corresponding computations into smaller units. Both techniques apply in other distributed frameworks, such as Corba, Java RMI.

Specifically, in the framework of Java, we have defined and implemented the ProActive library [18] that offers a similar model with Java and its virtual machine. The framework being much more dynamic there are several new features that we are able to provide. First of all, it is not only possible to create remotely accessible objects, but also to "*turn active*" an existing object. Within the context of dynamic class loading, a JVM can receive an object for which the class was previously unknown, and the library make it possible to make this object remotely accessible. Another strong new feature is the *mobility of computations*. A migration primitive allows an active object to move from one machine to another, while maintaining functional all the remote references to and from itself. The graphical environment IC2D (Interactive Control and Debugging of Distribution) makes it possible to monitor and steer distributed and parallel applications. In Figure 15, the machines, the JVMs, and the active objects are graphically represented, as well as the communications that take place. Moreover, *drag*

*and drop migration* allows to move around objects at execution, potentially from one continent to another in a metacomputing framework.

Finally, another important aspect of distributed programming is the striking correctness problems it raises. This paper didn't address them, but it is another area of investigation for our group [4, 5]. We hope formal techniques, together with parallel object-oriented programming, will permit some advances in that matter.

## References


[1] *Parallel Virtual Machine: a user's guide and tutorial for networked parallel computing*. MIT Press (1994).

[2] *MPI: The Complete Reference*. MIT Press (1998).

[3] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press (1986).

[4] I. Attali, D. Caromel and M. Oudshoorn, A Formal Definition of the Dynamic Semantics of the Eiffel Language, in *Sixteenth Australian Computer Science Conference (ACSC-16)*, G. Gupta, G. Mohay and R. Topor Eds., Griffith University, February (1993) 109–120.

[5] I. Attali, D. Caromel and M. Russo, Graphical Visualization of Java Objects, Threads, and Locks. *IEEE Distributed Systems Online* **2** (2001).

[6] H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum and J. Jansen, Replication techniques for speeding up parallel applications on distributed systems. *Concurrency Practice & Experience* **4** (1992) 337–355.

[7] F. Baude, D. Caromel, N. Furmento and D. Sagnol, Optimizing Metacomputing with Communication-Computation Overlap, in *6th International Conference PaCT 2001*, number 2127, V. Malyshkin Ed., LNCS, 190–204.

[8] A. Birrell, G. Nelson, S. Owicki and E. Wobber, Network Objects. Technical Report SRC-RR-115, DEC Systems Research Center (1995).

[9] G. Booch, Object-Oriented Development. *IEEE Transaction on Software Engineering* (1986).

[10] T. Brandes and F. Desprez, Implementing Pipelined Computation and Communication in an HPF Compiler, in *Euro-Par'96*, number 1123, LNCS.

[11] F. Buschmann, K. Kiefer, F. Paulish and M. Stal, The Meta-Information-Protocol: Run-Time Type Information for C++, in *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, A. Yonezawa and B.C. Smith Eds. (1992) 82–87.

[12] D. Caromel, Service, Asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming* **2** (1989) 12–22.

[13] D. Caromel, Concurrency: an Object Oriented Approach, in *Technology of Object-Oriented Languages and Systems (TOOLS'90)*, J. Bezivin, B. Meyer and J.-M. Nerson Eds., Angkor, June (1990) 183–197.

[14] D. Caromel, Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming* **3** (1990) 34–42.

[15] D. Caromel, Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM* **36** (1993) 90–102.

[16] D. Caromel, F. Belloncle and Y. Roudier, The C++// Language, in *Parallel Programming Using C++*, MIT Press (1996) 257–296.

[17] D. Caromel, P. Dzwig, R. Kauffman, H. Liddell, A. McEwan, P. Mussi, J. Poole, M. Rigg and R. Winder, EC++ – EUROPA Parallel C++: A Draft Definition, in *Proceedings of High-Performance Computing and Networking (HPCN'96)*, Vol. 1067, LNCS, 848–857.

[18] D. Caromel, W. Klauser and J. Vayssiere, Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience* (1998).

[19] D. Caromel, E. Noulard and D. Sagnol, Sharedonread optimization in parallel object-oriented programming, in *Computing in Object-Oriented Parallel Environments, Proceedings of ISCOPE'99*, LNCS, San Francisco, Dec (1999).

[20] S. Chiba and T. Masuda, Designing an Extensible Distributed Language with Meta-Level Architecture, in *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, O. Nierstrasz Ed., Springer-Verlag, Kaiserslautern, *Lecture Notes in Computer Science* **707** (1993) 482–501.

[21] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R.C. Whaley, A proposal for a set of parallel basic linear algebra subprograms. Technical Report Lapack Working Note 100, May (1995).

[22] A. Dave, M. Sefika and R.H. Campbell, Proxies, Application Interfaces and Distributed Systems, in *proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems (OOOS), Paris (France)*, IEEE Computer Society Press, September (1992).

[23] F. Desprez, P. Ramet and J. Roman, Optimal Grain Size Computation for Pipelined Algorithms, in *Euro-Par'96*, number 1123, LNCS.

[24] S. Diwan and D. Gannon, Capabilities Based Communication Model for High-Performance Distributed Applications: The Open HPC++ Approach, in *IPPS/SPDP* (1999). `ftp://ftp.cs.indiana.edu/pub/sdiwan/capab.ps.gz`

[25] I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* **11** (1997) 115–128.

[26] D. Gannon, S. Diwan and E. Johnson, HPC++ and the Europa Call Reification Model. *ACM Applied Computing Review* **4** (1996).
[27] N. Gehani, Concurrent Programming in the ADA Language: the Polling Bias. *Software–Practice and Experience* **14** (1984).
[28] R. Halstead, Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October (1985).
[29] S.B. Hassen and H. Bal, Integrating task and data parallelism using shared objects, in *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, PA, USA, May 25–28, 1996*, ACM Ed., ACM Press, New York (1996) 317–324.
[30] C. Hewitt, Viewing Control Structures as Patterns of Passing Messages. *J. Artificial Intelligence Res.* **8** (1977) 323–64.
[31] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda and K. Kubota, Design and implementation of metalevel architecture in C++ - MPC++ approach, in *Reflection'96*, April (1996).
[32] G. Kiczales, J. des Rivières and D.G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press (1991).
[33] H. Lieberman, Concurrent Object-Oriented Programming in Act 1, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).
[34] P. Madany, N. Islam, P. Kougiouris and R.H. Campbell, Practical Examples of Reification and Reflection in C++, in *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, A. Yonezawa and B.C. Smith Eds. (1992) 76–81.
[35] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall (1988).
[36] E. Noulard, N. Emad and L. Flandrin, Calcul numérique parallèle et technologies objet. Technical Report Rapport PRISM 1998/003, ADULIS/PRiSM, Juillet (1997). Révision du 30/01/98.
[37] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr and R. Sanzi, Mach: a foundation for open systems (operating systems), in *Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II), Pacific Grove, CA, USA, September 27–29, 1989*, IEEE Ed., IEEE Computer Society Presspages (1989) 109–113.
[38] Y. Saad, *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, New York (1996).
[39] M. Shapiro, Structure and Encapsulation in Distributed Systems: the Proxy Principle, in *Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, MA, USA*, IEEE, May (1986) 198–204.
[40] C.W. Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. Ph.D. thesis, Rice University (1993).
[41] G. Wilson and P. Lu Eds., *Parallel Programming Using C++*. MIT Press (1996).
[42] Y. Yokote and M. Tokoro, Concurrent Programming in ConcurrentSmalltalk, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).
[43] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro Eds., MIT Press (1987).

### 2.2.2 Programming, Composing, Deploying for the Grid.

L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid (chapter 9). Springer, 2006. Pages 205–229. ISBN : 1-85233-998-5.

# Programming, Composing, Deploying for the Grid

Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici

OASIS - Joint Project CNRS / INRIA / University of Nice Sophia-Antipolis
INRIA - 2004 route des Lucioles - B.P. 93 - 06902 Valbonne Cedex, France
FirstName.LastName@sophia.inria.fr

**Abstract.** Grids raise new challenges in the following way: heterogeneity of underlying machines/networks and runtime environments (types and performance characteristics), not a single administrative domain, versatility. So the need to have appropriate programming and runtime solutions in order to write, deploy then execute applications on such heterogeneous distributed hardware in an effective and efficient manner. We propose in this article a solution to those challenges which takes the form of a programming and deployment framework featuring parallel, mobile, secure and distributed objects and components.
Keywords: Distributed objects, components, mobility, monitoring, deployment, security, mapping.

## 1 Introduction

### 1.1 Motivation

In this article, we present a contribution to the problem of software reuse, integration and deployment for parallel and distributed computing. Our approach takes the form of a programming and deployment framework featuring parallel, mobile, secure and distributed objects and components. We especially target Grid computing, but our approach also applies to application domains such as mobile and ubiquitous distributed computing on the Internet (where high performance, high availability, ease of use, etc., are of importance).

Below are the main current problems raised by grid computing that we have identified, and for which we provide some solutions. For Grid applications development, there is a need to smoothly, seamlessly and dynamically integrate and deploy autonomous software, and for this to provide a *glue* in the form of a software bus. Additionally, complexification of distributed applications and commodity of resources through grids are making the tasks of deploying those applications harder. So, there is a clear need for standard tools allowing versatile deployment and analysis of distributed applications. Grid applications must be able to cope with large variations in deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. As a consequence, the security should not be tied up in the application code, but rather

easily configurable in a flexible, and abstract manner. Moreover, any large scale Grid application using hundreds or thousands of nodes will have to cope with migration of computations, for the sake of load balancing, change in resource availability, or just node failures.

We propose programming concepts, methodologies, and a framework for building meta-computing applications, that we think are well adapted to the hierarchical, highly distributed, highly heterogeneous nature of grid-computing. The framework is called *ProActive*, a Java-based middleware (programming model and environment) for object and component oriented parallel, mobile and distributed computing. As this article will show, *ProActive* is relevant for grid computing due to its secure deployment and monitoring aspects [1, 2], its efficient and typed collective communications [3], and component-based programming facilities [4] thanks to an implementation of the Fractal component model [5, 6], taking advantage of its hierarchical approach to component programming.

### 1.2 Context

Grid programmers may be categorized into three groups, such as defined in [7]: the first group are end users who program pre-packaged Grid applications by using a simple graphical or Web interface; the second group are those that know how to build Grid applications by composing them from existing application "components", for instance by programming (using scripting or compiled languages); the third group consists of the developers that build the individual components. Providing the user view of the Grid can also be seen as a two-levels programming model [8]: the second level is the integration of distributed components (developped at the first level), together into a complete *executable*.

In this context, the component model we propose addresses the second group of programmers; but we also address the third group by proposing a deployment and object-oriented programming model for *autonomous* grid-aware distributed software that may further be integrated if needed.

In the context of object oriented computing for grid, for which security is a concern, works such as Legion [9, 10] also provide an integrated approach like we do. But the next generation of programming models for wide area distributed computing is aimed at further enforcing code reuse and simplifying the developer's and integrator's task, by applying the component oriented methodology. We share the goal of providing a component-based high-performance computing solution with several projects such as: CCA [7] with the CCAT/XCAT toolkit [11] and Ccaffeine framework, Parallel CORBA objects [12] and GridCCM [13]. But, to our knowledge, what we propose is the first framework featuring *hierarchical* distributed components. This should clearly help in mastering the complexity of composition, deployment, re-usability required when programming and running large-scale distributed applications.

### 1.3 Plan

The organization of the paper is as follows: first we describe the parallel and distributed programming model we advocate for developing autonomous grid-aware software. Second, we define the concept of hierarchical, parallel, and distributed components yielding the concept of *Grid components*. The third part of the paper deals with concepts and tools useful during the life-cycle of a Grid application: deployment that moreover might need to be secure, visualization and monitoring of a Grid application and its associated runtime support (e.g. Java Virtual Machines), re-deployment by moving running activities and by rebuilding components.

## 2 Programming Distributed Mobile Objects

### 2.1 Motivation

As grid computing is just one particular instance of the distributed computing arena, our claim is that proposed programming models and tools should not drastically depart from traditional distributed computing.

We present here the parallel and distributed conceptual programming model and at the same time, one of the many implementations we have done, called *ProActive*, which is in Java, besides others done with Eiffel and C++, resp. called *Eiffel//* and *C++//* [14, 15]. The choice of the Java language is fundamental in order to hide heterogeneity of runtime supports, while the performance penalty is not too high compared with native code implementations (c.f. Java Grande benchmarks for instance [16]).

As *ProActive* is built on top of the Java standard APIs, mainly Java RMI and the Reflection APIs, it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified virtual machine. Additionally, the Java platform provides dynamic code loading facilities, very useful for tackling with complex deployment scenarios.

### 2.2 Base Model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [17]. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee. All of this semantics is built using meta programming techniques, which provide transparency and

the ground for adaptation of non-functional features of active objects to various needs.

We have deliberately chosen not to use an explicit message-passing based approach: we aim at enforcing code reuse by applying the remote method invocation pattern, instead of explicit message-passing.

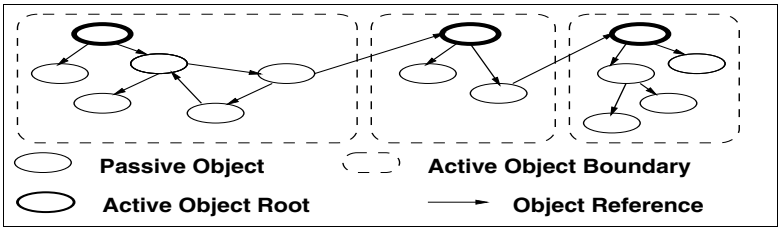



**Fig. 1.** A typical object graph with active objects

Another extra service (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For the sake of generality and dynamicity, creations of remotely accessible objects are triggered entirely at runtime by the application; nevertheless, active objects previously created and registered within another application may be accessed by the application, by first acquiring them with a lookup operation.

For that reason, JVMs need to be identified and added a few services. *Nodes* provide those extra capabilities : a *node* is an object defined in the model whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance `rmi://lo.inria.fr/Node`.

Let us consider a standard Java class `A`:

```
class A {
  public A() {}
  public void foo (...) {...}
  public V bar (...) {...}
   ...
}
```

The instruction:

```
A a = (A) ProActive.newActive("A",params,"//lo.inria.fr/Node");
```

creates a new active object of type `A` on the JVM identified with `Node`. It assumes that `Node` (i.e. the JVM) has been already deployed (see below and also section

4). Note that an active object can also be bound dynamically to a node as the result of a migration. Further, all calls to that remote object will be asynchronous, and subject to the *wait-by-necessity*:

```
a.foo (...);     // Asynchronous call
v = a.bar (...); // Asynchronous call
 ...
v.f (...);       // Wait-by-necessity: wait until v gets its value
```

### 2.3 Mobility

*ProActive* provides a way to move an active object from any Java virtual machine to any other one. This feature is accessible through a simple `migrateTo(...)` primitive (see Table 1).

| | Migration towards: |
|---|---|
| `migrateTo (URL)` | a VM identified by a URL |
| `migrateTo (Objet)` | the location of another Active Object |

**Table 1.** Migration primitives in *ProActive*

The primitive is `static`, and as such always triggers the migration of the current active object and all its passive objects. However, as the method is `public`, other, possibly remote, active objects can trigger the migration; indeed, the primitive implements what is usually called *weak migration*. The code in Example 2 presents such a mobile active object.

```
public class SimpleAgent implements Serializable {
  public void moveToHost(String t) {
      ProActive.migrateTo(t);
  }
  public void joinFriend(Object friend) {
      ProActive.migrateTo(friend);
  }
  public ReturnType foo(CallType p) {
      ...
  }
}
```

**Fig. 2.** SimpleAgent

In order to ensure the working of an application in the presence of migration, we provide three mechanism to maintain communication with mobile objects.

The first one relies on a location sever which keeps track of the mobile objects in the system. When needed, the server is queried to obtain an up-to-date reference to an active object. After migrating, an object updates its new location.

The second one uses a fully decentralized technique known as *forwarders* [18]. When leaving a site, an active object leaves a special object called a forwarder which points to its new location. Upon receiving a message, a forwarder simply passes it to the object (or another forwarder).

The third one is an original scheme based on a mix between forwarding an location server which provides both performance and fault tolerance.

### 2.4 Group Communications

The group communication mechanism of *ProActive* achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies.

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remain typed. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.

Here is an example of a typical group creation, based on the standard Java class `A` presented above:

```
   // A group of type "A"  and its 2 members are created at once
   // on the nodes directly specified,
   //  parameters are specified in params,
Object[][] params = {{...}, {...}};
Node[] nodes = {..., ...};
A ag = (A) ProActiveGroup.newActiveGroup("A", params, nodes);
```

Elements can be included into a typed group only if their class equals or extends the class specified at the group creation. Note that we do allow and handle *polymorphic* groups. For example, an object of class `B` (`B` extending `A`) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

A method invocation on a group has a syntax similar to that of a standard method invocation:

```
ag.foo(...); // A group communication
```

Such a call is asynchronously propagated to all members of the group using multi-threading. Like in the *ProActive* basic model, a method call on a group is non-blocking and provides a transparent future object to collect the results. A method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be a *ProActive* call and if the member is a standard Java object, the method call will be a standard Java

method call (within the same JVM). The parameters of the invoked method are broadcasted to all the members of the group.

An important specificity of the group mechanism is: the *result* of a typed group communication *is also a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited the caller blocks, but as soon as one reply arrives in the result group, the method call on this result is executed. For instance in:

```
V vg = ag.bar(); // A method call on a group, returning a result
                 // vg is a typed group of "V"
vg.f();          // This is also a collective operation
```

a new `f()` method call is automatically triggered as soon as a reply from the call `ag.bar()` comes back in the group `vg` (dynamically formed). The instruction `vg.f()` completes when `f()` has been called on all members.

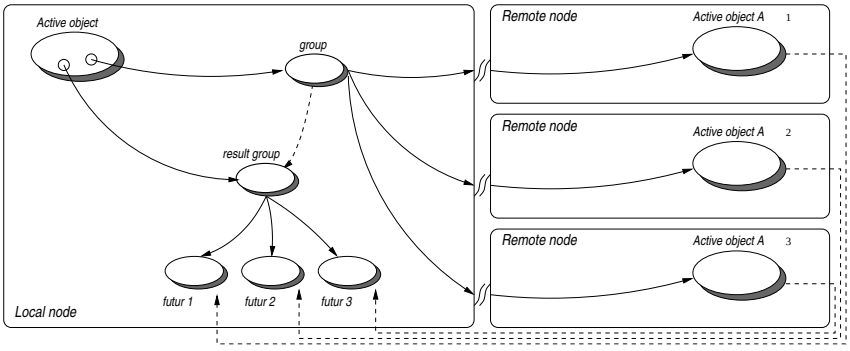



**Fig. 3.** Execution of an asynchronous and remote method call on group with dynamic generation of a result group

Other features are available regarding group communications: parameter dispatching using groups (through the definition of *scatter* groups), hierarchical groups, dynamic group manipulation (`add`, `remove` of members), group synchronization and barriers (`waitOne`, `waitAll`, `waitAndGet`); see [3] for further details and implementation techniques.

### 2.5 Abstracting Away from the Mapping of Active Objects to JVMs: Virtual Nodes

Active objects will eventually be deployed on very heterogeneous environments where security policies may differ from place to place, where computing and

communication performances may vary from one host to the other, etc. As such, the effective locations of active objects must not be tied in the source code.

A first principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols,

the goal being to deploy any application anywhere without changing the source code. For instance, we must be able to use various protocols, `rsh, ssh, Globus, LSF,` etc., for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as `RMIregistry, Jini, Globus, LDAP, UDDI,` etc. Therefore, we see that the creation, registration and discovery of resources has to be done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. The description should indicate the various parallel or distributed entities in the program or in the component. As we are in a (object-oriented) message passing model, to some extend, this description indicates the maximum number of address spaces. For instance, an application that is designed to use three interactive visualization nodes, a node to capture input from a physic experiment, and a simulation engine designed to run on a cluster of machines should somewhere clearly advertise this information.

Now, one should note that the abstract description of an application and the way to deploy it are not independent piece of information. In the example just above, if there is a simulation engine, it might register in a specific registry protocol, and if so, the other entities of the computation might have to use that lookup protocol to bind to the engine. Moreover, one part of the program can just lookup for the engine (assuming it is started independently), or explicitly create the engine itself.

To summarize, in order to abstract away the underlying execution platform, and to allow a *source-independent deployment*, a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

Besides the principles above, we want to eliminate as much as possible the use of scripting languages, that can sometimes become even more complex than application code. Instead, we are seeking a solution with XML descriptors, XML editor tools, interactive ad-hoc environments to produce, compose, and activate descriptors (see section 4).

To reach that goal, the programming model relies on the specific notion of **Virtual Nodes** (VNs):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN is defined and configured in a deployment descriptor (XML) (see section 4 for further details),
- a VN, after activation, is mapped to one or to a set of *actual* ProActive *Nodes*.

Of course, distributed entities (active objects), are created on Nodes, not on Virtual Nodes. There is a strong need for both Nodes and *Virtual Nodes*. *Virtual Nodes* are a much richer abstraction, as they provide mechanisms such as *set* or *cyclic mapping*. Another key aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical if we want to get at once machines from a cluster of PCs managed through Globus or LSF. It is even more critical in a Grid application, when trying to achieve the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a deployment descriptor (see section 4):

- the mapping of VNs to Nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup VNs.

```
ProActiveDescriptor pad =
            ProActive.getProActiveDescriptor(String xmlFileLocation);
//---- Returns a ProActiveDescriptor object from the xml file
VirtualNode dispatcher = pad.getVirtualNode("Dispatcher");
//---- Returns the VirtualNode Dispatcher described
//     in the xml file as a java object
dispatcher.activate()
// --- Activates the VirtualNode
Node node = dispatcher.getNode();
//-----Returns the first node available among nodes mapped
//     to the VirtualNode
C3DDispatcher c3dDispatcher = newActive("C3DDispatcher", param, node);
.........................
```

**Fig. 4.** Example of a *ProActive* source code for descriptor-based mapping

Now, within the source code, the programmer can manage the creation of active objects without relying on machine names and protocols. For instance, the piece of code given in Figure 4 will allow to create an active object onto the Virtual Node `Dispatcher`. The Nodes (JVMs) associated in a descriptor file with a given VN are started (or acquired) only upon activation of a VN mapping (`dispatcher.activate()` in the Figure 4).

## 3 Composing

### 3.1 Motivation

The aim of our work around components is to combine the benefits of a component model with the features of *ProActive*. The resulting components, that we call "Grid components", are recursively formed of either sequential, parallel and/or distributed sub-components, that may wrap legacy code if needed, and that may be deployed but further reconfigured and moved – for example to tackle fault-tolerance, load-balancing or adaptability to changing environmental conditions.

Here is a typical scenario illustrating the usefulness of our work. Consider complex grid software formed of several services, say of other software (a parallel and distributed solver, a graphical 3D renderer, etc). The design of this grid software is highly simplified if it can be considered as a hierarchical composition (recursive assembly and binding): the solver is itself a component composed of several components, each one encompassing a piece of the computation. The whole software is seen as a single component formed of the solver and the renderer. From the outside, the usage of this software is as simple as invoking a functional service of a component (e.g. call *solve-and-render*). Once deployed and running on a grid, assume that due to load balancing purposes, this software needs to be relocated. Some of the ongoing computations may just be moved (the solver for instance); others depending on specific peripherals that may not be present at the new location (the renderer for instance) may be terminated and replaced by a new instance adapted to the target environment and offering the same service. As the solver is itself a hierarchical component formed of several sub-components, each encompassing an activity, we trigger the migration of the solver as a whole, without having to explicitly move each of its sub-components, while references towards mobile components remain valid. Eventually, once the new graphical renderer is launched, we re-bind the software, so as it now uses this new configuration.

### 3.2 Component Model

Observing the works done so far on component software, including standardized industrial component models, such as CCM, EJB or COM, some researchers concluded that there was still missing an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. They

consequently introduced a new model[5], based on the concepts of encapsulation (components are black boxes), composition (the model is hierarchical), sharing[1], life-cycle (a component lives through different phases), activities, control (this allows the management of non-functional properties of the components), and dynamicity (this allows reconfiguration). This model is named Fractal.

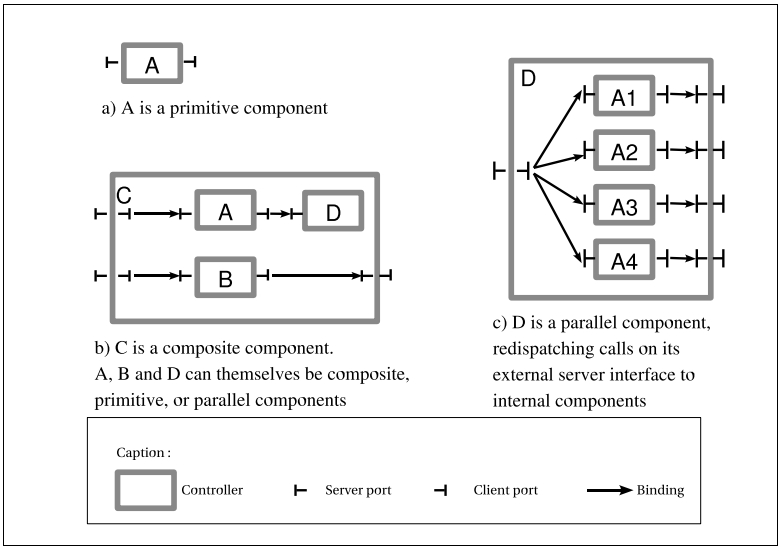



**Fig. 5.** The 3 types of components

The Fractal model is somewhat inspired from biological cells, i.e. plasma surrounded by membranes. In other words, a component is formed out of two parts: a *content*, and a set of *controllers*. The content can be recursive, as a component can contain other components: the model is hierarchical. The controllers provide introspection capabilities for monitoring and exercising control over the execution of the components. A component interacts with its environment (notably, other components) through well-defined *interfaces*. These interfaces can be either client or server, and are interconnected using *bindings* (see fig. 5).

Fractal is a component model conceived to be simple but extensible. It provides an API in Java, and offers a reference implementation called Julia. Unfortunately, Julia is not based on a distributed communication protocol (although there exists a Jonathan personality, i.e. a set of RMI Fractal components), thus hindering the building of systems with distributed components.

[1] sharing is currently not supported in the *ProActive* implementation

Besides, *ProActive* offers many features, such as distribution, asynchronism, mobility or security, that would be of interest for Fractal components.

We therefore decided to write a new implementation of the Fractal API based on *ProActive*, that would benefit from both sides, and that would ease the construction of distributed and complex systems.

### 3.3 *ProActive* Components

A *ProActive* component has to be parallelizable and distributable as we aim at building grid-enabled applications by hierarchical composition; componentization acts as a glue to couple codes that may be parallel and distributed codes requiring high performance computing resources. Hence, components should be able to encompass more than one activity and be deployed on parallel and distributed infrastructures. Such requirements for a component are summarized by the concept we have named *Grid Component*.

Figure 5 summarizes the three different cases for the structure of a Grid component as we have defined it. For a composite built up as a collection of components providing common services (Figure 5.c), *group communications* (see 2.4) are essential for ease of programming and efficiency. Because we target high performance grid computing, it is also very important to efficiently implement point-to-point and group method invocations, to manage the deployment complexity of components distributed all over the Grid and to possibly debug, monitor and reconfigure the running components.

A synthetic definition of a *ProActive* component is the following :

- It is formed from one (or several) Active Object(s), executing on one (or several) JVM(s)
- It provides a set of server ports (Java Interfaces)
- It possibly defines a set of client ports (Java attributes if the component is primitive)
- It can be of three different types :
  1. primitive : defined with Java code implementing provided server interfaces, and specifying the mechanism of client bindings.
  2. composite : containing other components.
  3. parallel : also a composite, but re-dispatching calls to its external server interfaces towards its inner components.
- It communicates with other components through 1-to-1 or group communications.

A *ProActive* component can be configured using :

- an XML descriptor (defining use/provide ports, containment and bindings in an Architecture Description Language style)
- the notion of virtual node, capturing the deployment capacities and needs

Finally, we are currently working on the design of specialized components encapsulating legacy parallel code (usually Fortran-MPI or C-MPI). This way, *ProActive* will allow transparent collaboration between such legacy applications and any other Grid component.

### 3.4 Example

We hereby show an example of how a distributed component system could be built using our component model implementation. It relates to the scenario exposed in section 3.1.

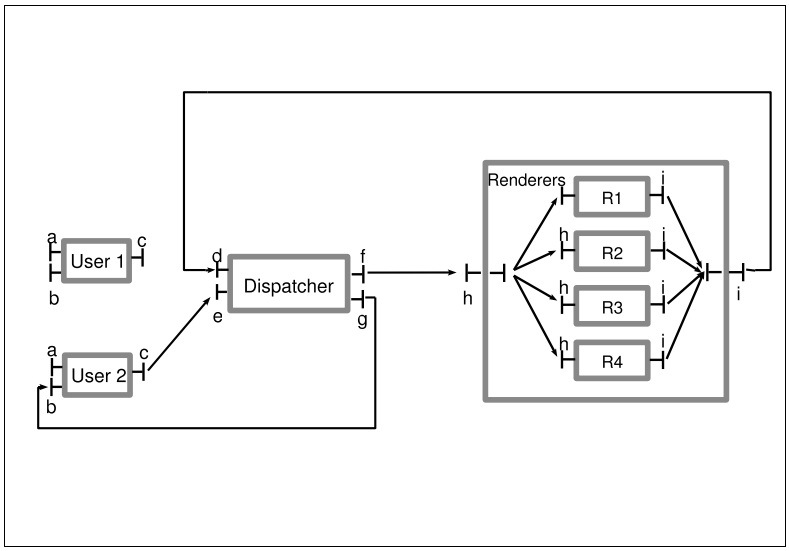



**Fig. 6.** A simplified representation of the C3D component model

C3D, an existing application, is both a collaborative application and a distributed raytracer: users can interact through messaging and voting facilities in order to choose a 3D scene that is rendered using a set of distributed rendering engines working in parallel. This application is particularly suitable for component programming, as we can distinguish individual software entities and we can abstract client and server interfaces from these entities. The resulting component system is shown in figure 6 : users interact with the dispatcher component, can ask for the scene motion, and can see the evolution of the ray-tracing. The dispatcher delegates the calculation of the scene to a parallel component (renderers). This parallel component contains a set of rendering engines (R1, R2, R3), and distributes calculation units to these rendering engines thanks to the

group communication API (scatter feature, see 2.4). The results are then forwarded (from the client interface `i` of the renderers component) as a call-back to the dispatcher (server interface `d`), and later to the users (from client interface `g` of the dispatcher to server interface `b` of the clients). These relations result in cyclic composition of the components. During the execution, users (for instance user1 represented on the figure) can dynamically connect to the dispatcher and interact with the program. Another dynamical facility is the connection of new rendering engine components at runtime : to speedup the calculations, if the initial configuration does not perform fast enough, new rendering engines can be added along with R1, R2, R3. Besides, components being active objects, they can also migrate for load-balancing or change of display purposes, either programmatically or interactively using tools such as IC2D (see 4.3). Interaction between users, like votes, is not represented here.

There are two ways of configuring and instantiating component systems : either programmatically, or using an architecture description language (ADL). The ADL can help a lot, as it automates the instantiation, the deployment, the assembly and the binding of the components. The following examples correspond to the configuration of the C3D application. The ADL is composed of two main sections. The first section defines the types of the components (User-Type, Dispatcher-Type and Renderer-Type), in other words the services the components offer and the services they require :

```
<types>
    <component-type name="User-Type">
        <provides>
            <interface name="a" signature="package.UserInput"/>
            <interface name="b" signature="package.SceneUpdate"/>
        </provides>
        <requires>
            <interface name="c" signature="package.UserSceneModification"/>
        </requires>
    </component-type>
    <component-type name="Dispatcher-Type">
        <provides>
            <interface name="d" signature="package.UserSceneModification"/>
            <interface name="e" signature="package.CalculationResult"/>
        </provides>
        <requires>
            <interface name="f" signature="package.CalculateScene"/>
            <interface name="g" signature="package.SceneUpdate"/>
        </requires>
    </component-type>
    <component-type name="Renderer-Type">
        <provides>
            <interface name="h" signature="package.Rendering"/>
        </provides>
        <requires>
            <interface name="i" signature="package.CalculationResult"/>
```

```
        </requires>
    </component-type>
</types>
```

The second section defines the instances of the components, the assembly of components into composites, and the bindings between components :

```
<components>
    <primitive-component implementation="package.User"
                    name="user1" type="User-Type"
                    virtualNode="UserVN"/>
    <primitive-component implementation="package.User"
                    name="user2" type="User-Type"
                    virtualNode="UserVN"/>
    <primitive-component implementation="package.Dispatcher"
                    name="dispatcher" type="Dispatcher-Type"
                    virtualNode="DispatcherVN"/>
    <parallel-component name="parallel-renderers"
                    type="Renderer-Type"
                    virtualNode="parallel-renderers-VN">
        <components>
            <primitive-component implementation="package.Renderer"
                    name="renderer" type="Renderer-Type"
                    virtualNode="renderers-VN"/>
            <!-- the actual number of renderer instances
            depends upon the mapping of the virtual node -->
        </components>
        <!-- bindings are automatically performed
        inside parallel components -->
    </parallel-component>
</components>
<bindings>
    <binding client="dispatcher.c" server="parallel-renderers.c"/>
    <binding client="renderers.r" server="dispatcher.r"/>
    <binding client="user1.i" server="dispatcher.i"/>
    <binding client="dispatcher.g" server="user2.b"/>
    <!-- bindings to clients can also be performed dynamically
    as they appear once the application is started
    and ready to receive input operations -->
</bindings>
```

Bindings connect components at each level of the hierarchy, and are performed automatically inside parallel components. The primitive components contain functional code from the class specified in the implementation attribute.

Each component also exhibits a "virtual node" property : the design of the component architecture is decoupled from the deployment (see 4.2) of the components. This way, the same component system can be deployed on different computer infrastructures (LAN, cluster, Grid).

In conclusion, the benefits of the componentization of the C3D application are – at least – threefold. First, the application is easier to understand and to

configure. Second, the application is more evolutive: for instance, as the rendering calculations are encapsulated in components, one could improve the rendering algorithm, create new rendering engine components and easily replace the old components with the new ones. Third, the application is easier to deploy, thanks to the mapping of the components onto virtual nodes.

## 4 Deploying, Monitoring

### 4.1 Motivation

Increasing complexity of distributed applications and commodity of resources through grids are making the tasks of deploying those applications harder. There is a clear need for standard tools allowing versatile deployment and analysis of distributed applications. We present here concepts for the deployment and monitoring, and their implementation as effective tools integrated within the *ProActive* framework. If libraries for parallel and distributed application development exist (RMI in Java, jmpi [19] for MPI programming, etc.) there is no standard yet for the deployment of such applications. The deployment is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers, clusters or grids. The commoditization of resources through grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

Questions such as "are the distributed entities correctly created ?", "do the communications among such entities correctly execute ?", "where is a given mobile entity actually located ?", etc. are usually left unanswered. Moreover, there is usually no mean to dynamically modify the execution environment once the application is started. Grid programming is about deploying processes (activities) on various machines. In the end, the security policy that must be ensured for those processes depends upon many factors: first of all, the application policy that is needed, but also, the machine locations, the security policies of their administrative domain, and the network being used to reach those machines.

Clearly said, the management of the mapping of processes (such as JVMs, PVM or MPI daemons) onto hosts, the deployment of activities onto those processes have generally to be explicitly taken into account, in a static way, sometimes inside the application, sometimes through scripts. The application cannot be seamlessly deployed on different runtime environments.

To solve those critical problems, the quite classical and somehow ideal solutions we propose follow 4 steps:

1. abstract away from the hardware and software runtime configuration by introducing and manipulating in the program virtual processes where the activities of the application will be subsequently deployed,
2. provide external information regarding all real processes that must be launched and the way to do it (it can be through remote shells or job submission to clusters or grids), and define the mapping of virtual processes onto real processes,

3. provide a mean to visualize, complete or modify the deployment once the application has started,
4. provide an infrastructure where Grid security is expressed outside the application code, outside the firewall of security domains, and in both cases in a high-level and flexible language.

### 4.2 Deployment Descriptors

We solve the two first steps by introducing XML-based descriptors able to describe activities and their mapping onto processes. Deployment descriptors allow to describe: (1) virtual nodes, entities manipulated in the source code, representing containers of activities, (2) Java virtual machines where the activities will run and the way to launch or find them, (3) the mapping between the virtual nodes and the JVMs. The deployment of the activities is consequently separated from the code; one can decide to deploy the application on different hosts just by adapting the deployment descriptor, without any change to the source code.

Descriptors are structured as follows:

```
virtual nodes
        definition
        acquisition
deployment
        security
        register
        lookup
        mapping
        jvms
infrastructure
```

**Virtual Nodes** As previously stated (see 2.5), a virtual node is a mean to define a mapping between a conceptual architecture and one or several nodes (JVMs), and its usage in the source code of a program has been given on figure 4.

The names of the virtual nodes in the source code has to correspond to the names of the virtual nodes defined in the first section. There are two ways of using virtual nodes. The first way is to name them (and further explicitly describe them):

```
<virtualNodesDefinition>
    <virtualNode name="User"/>
</virtualNodesDefinition>
```

The second way is to acquire virtual nodes already deployed by another application:

```
<virtualNodesAcquisition>
    <virtualNode name="Dispatcher"/>
</virtualNodesAcquisition>
```

**Deployment** The deployment section defines the correspondence, or mapping, between the virtual nodes in the first section, and the processes they actually create.

The security section allows for the inclusion of security policies in the deployment (see section 4.4):

```
<security file="URL">
```

The register section allows for the registration of virtual nodes in a registry such as `RMIRegistry` or `JINI lookup service`:

```
<register virtualNode="Dispatcher" protocol="rmi">
```

This way, the virtual node "Dispatcher", as well as all the JVMs it is mapped to, will be accessible by another application through the rmi registry.

Symmetrically, descriptors provide the ability to acquire a virtual node already deployed by another application, and defined as acquirable in the first section:

```
<lookup virtualNode="Dispatcher" host="machineZ" protocol="rmi"/>
```

The mapping section helps defining:

- a one virtual node to one JVM mapping:

```
<map virtualNode="User1">
    <jvmSet>
        <currentJvm protocol="rmi"/>
        <!-- currentJvm is the Jvm of the file parsing process -->
    </jvmSet>
</map>
```

- a one virtual node to a set of JVMs mapping:

```
<map virtualNode="Renderer">
    <jvmSet>
        <vmName value=Jvm1/>
        <vmName value=Jvm2/>
        <vmName value=Jvm3/>
        ...
    </jvmSet>
</map>
```

- the collocation of virtual nodes, when two virtual nodes have a common mapping on a JVM.

Virtual nodes represent sets of JVMs, and these JVMs can be remotely created and referenced using standard Grid protocols. Each JVM is associated with a creation process, that is named here but fully defined in the "infrastructure" section. For example, here is an example where Jvm1 will be created through a Globus process that is only named here, but defined in the infrastructure section:

```
<jvms>
    <jvm name="Jvm1">
        <acquisition method="rmi"/>
        <creation>
            <processReference refid="GlobusProcess"/>
        </creation>
    </jvm>
    ...
</jvms>
```

**Infrastructure** The infrastructure section explicitly defines which Grid protocols (and associated *ProActive* classes) to use in order to create remote JVMs.

The remote creation of a JVM implies two steps: first, the connection to a remote host, second, the actual creation of the JVM.

Let us start with the second step. Once connected to the remote host, the JVM can be created using the `JVMNodeProcess` class:

```
<processDefinition id="jvmProcess">
    <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
<processReference refid="localJvmCreation"/>
```

The first step, the connection process, can itself invoke other processes. For example, the connection to LSF hosts requires beforehand a ssh connection to the frontal of the cluster, then a bSub command to reach the hosts inside the cluster. When connected, the previously defined `localJvmCreation` process is called:

```
<processDefinition id="bsubInriaCluster">
    <bsubProcess
        class="org.objectweb.proactive.core.process.lsf.LSFBSubProcess">
            <processReference refid="localJvmCreation"/>
            <bsubOption>
                <processor>20</processor>
            </bsubOption>
    </bsubProcess>
</processDefinition>
<processDefinition id="sshProcess">
    <sshProcess
        class="org.objectweb.proactive.core.process.ssh.SSHProcess"
        hostname="sea.inria.fr">
        <processReference refid="bsubInriaCluster"/>
    </sshProcess>
</processDefinition>
```

Other protocols are supported, including : rsh, rlogin, ssh, Globus, PBS. Here is an example using Globus:

```
<processDefinition id="globusProcess">
    <globusProcess
        class="org.objectweb.proactive.core.process.globus.GlobusProcess"
        hostname="globus.inria.fr">
        <processReference refid="localJvmCreation"/>
        <globusOption>
            <count>15</count>
        </globusOption>
    </globusProcess>
</processDefinition>
```

More information about the descriptors and how to use them is given in [20].

**Deployment of Components** The ADL used for describing component systems associates each component with a virtual node (see 3.4). A component system can be deployed using any deployment descriptor, provided the virtual node names match. The parallel components take advantage of the deployment descriptor in another way. In the example of section 3.4, consider the parallel component named "renderers". It only defines one inner component "renderer", and this inner component is associated to the virtual node "renderers-VN". If "renderers-VN" is mapped onto a single JVM A, only one instance of the renderer will be created, on the JVM A. But if this "renderers-VN" is actually mapped onto a set of JVMs, one instance of the renderer will be created on each of these JVMs. This allows for large scale parallelization in a transparent manner.

### 4.3 Interactive Tools

We solve the third step mentioned in section 4.1 by having a monitoring application: *IC2D* (Interactive Control and Debugging of Distribution). It is a graphical environment for monitoring and steering distributed *ProActive* applications.

**Monitoring the Infrastructure and the Mapping of Activities** Once a *ProActive* application is running, *IC2D* enables the user to graphically visualize fundamental distributed aspects such as topology and communications (see figure 7).

It also allows the user to control and modify the execution (e.g. the mapping of activities onto real processes, i.e. JVMs, either upon creation or upon migration. Indeed, it provides a way to interactively **drag-and-drop any running active object** to move it to any node displayed by *IC2D* (see figure 8). This is a useful feature in order to react to load unbalance, to expected unavailability of a host (especially useful in the context of a *desktop grid*), and more importantly in order to help implementing the concept of a *pervasive grid*: mobile users need

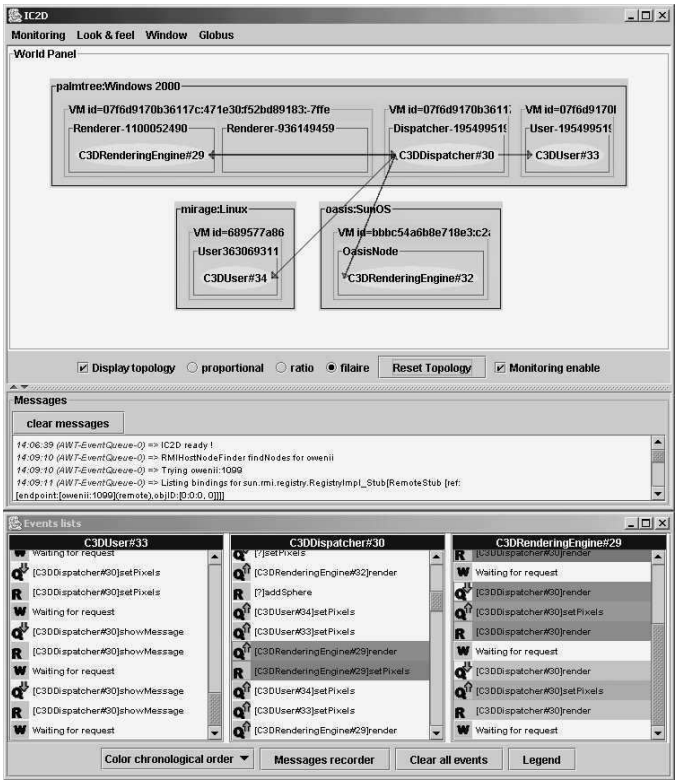



**Fig. 7.** General view of what *IC2D* displays when an application is running

to move the front-end active objects attached to the on-going grid computations they have launched, on their various computing devices so as to maintain their grid connectivity.

Moreover, it is possible to trigger the activation of new JVMs (see figure 9) adding dynamicity in the configuration and deployment.

**Interactive Dynamic Assembly and Deployment of Components** *IC2D compose* and *IC2D deploy* are two new interactive features we are planning to add to the *IC2D* environment.

The idea is to enable an integrator to graphically describe an application: the components, the inner components, and their respective bindings. The outcome of the usage of *IC2D compose* would be an automatically generated ADL. At this specific point, we need to provide the integrator with several solutions for **composing virtual nodes**. Indeed, as each *ProActive* component is attached to one virtual node, what about the virtual node of the composite component

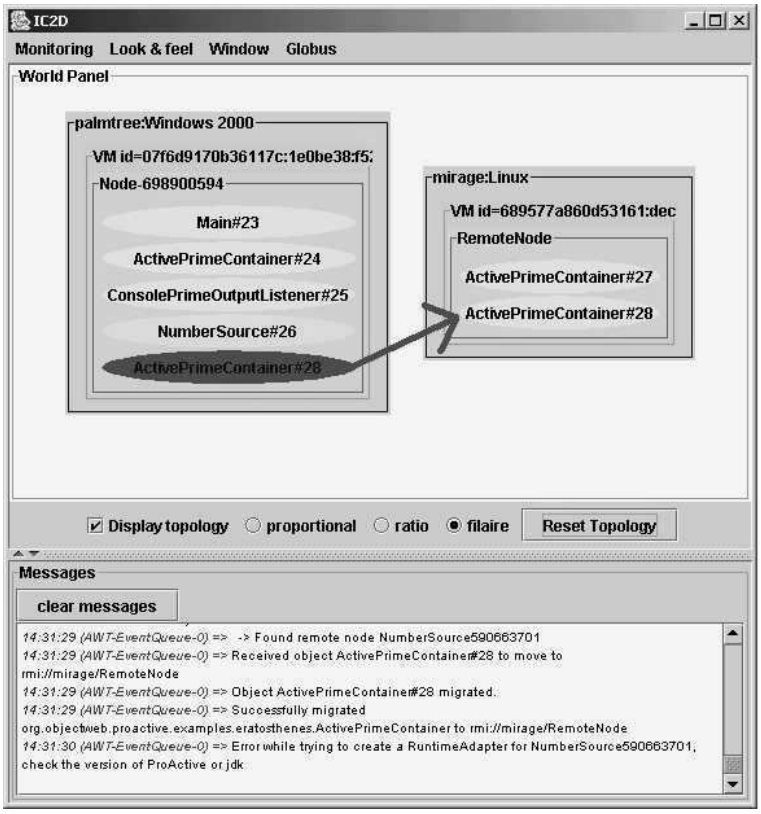



**Fig. 8.** Drag-and-drop migration of an active object

that results from the composition of several sub-components ? Should the resulting component still be deployed on the different virtual nodes of its inner components; or, on the contrary, should those virtual nodes be merged into one single virtual node attached to the composite ? The decision is grounded on the fact that merging virtual nodes is an application-oriented way to enforce the collocation of components.

Besides, *IC2D deploy* will be an additional tool that graphically would enable to trigger the deployment then the starting of an application based upon its ADL; this of course requires the complementary usage of a deployment descriptor attached to this application, so as to first launch the required infrastructure. Once the application has been deployed, *IC2D deploy* would enable to dynamically and graphically manage the life-cycle of the components (start, stop), and interface with *IC2D compose* so as to allow the user modify their bindings and

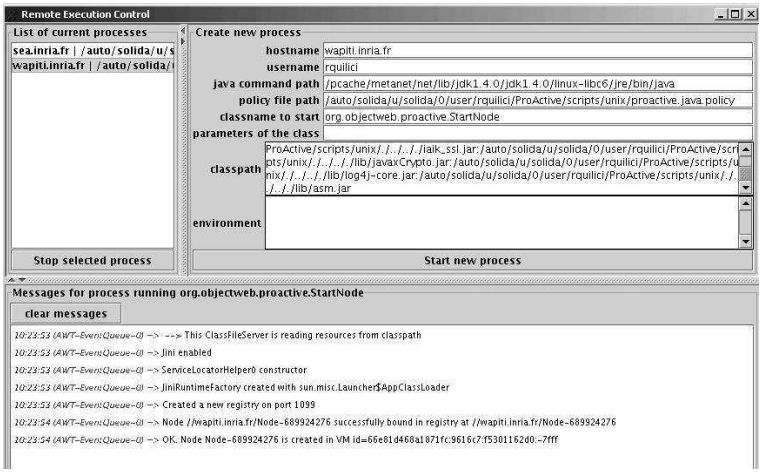



**Fig. 9.** Interactive creation of a new JVM and associated node

their inclusion. Of course, the *IC2D* monitor itself is still useful so as to visualize the underlying infrastructure and all activities. At this point, we will need to extend the *IC2D* monitor, so as to provide a way to graphically show all activities that are part of the same component: in this way, it will be possible to trigger the migration of a component as a whole, by using the usual drag-and-drop facility.

### 4.4 Security Enforcement

We now describe the fourth step mentioned in section 4.1. Grid applications must be able to cope with large variations in deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. In the same way as the deployment is not tied up in the source code, we provide a similar solution regarding the security, so as it becomes easily configurable in a flexible, and abstract manner. Overall, we propose a framework allowing:

- declarative security attributes (Authentication, Integrity, Confidentiality), outside any source code and away from any API;
- policies defined hierarchically at the level of administrative domain,
- dynamic security policy, taking into account the nature (private or public) of the underlying networks;
- dynamically negotiated policies (for multi-principals applications),
- policies for remote creation and migration of activities.

**A Hierarchical Approach to Grid Security** A first decisive feature allows to define application-level security on Virtual Nodes, those application-level deployment abstractions:

**Definition 1.** *Virtual Node Security*
*Security policies can be defined at the level of Virtual Nodes. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to JVMs, and Hosts.*

As such, virtual nodes are the support for intrinsic application level security. If, at design time, it appears that a process always requires a specific level of security (e.g. authenticated and encrypted communications at all time), then that process should be attached to a virtual node on which those security features are imposed. It is the designer responsibility to structure his/her application or components into virtual node abstractions compatible with the required security. Whatever deployment occurs, those security features will be maintained.

The second decisive feature deals with a major Grid aspect: deployment-specific security. The issue is actually twofold:

1. allowing organizations (security domains) to specify general security policies,
2. allowing application security to be specifically adapted to a given deployment environment.

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure; they are organized in a hierarchical manner. They are the logical concept allowing to express security policies in a hierarchical way.

**Definition 2.** *Declarative Domain Security*
*Fine grain and declarative security policies can be defined at the level of* Domains*. A Security Domain is a domain to which a certificate and a set of rules are associated.*

This principle allows to deal with the two issues mentioned above:
(1) the administrator of a domain can define specific policy rules that must be obeyed by the applications running within the domain. However, a general rule expressed inside a domain may prevent the deployment of a specific application. To solve this issue, a policy rule can allow a well-defined entity to weaken it. As we are in a hierarchical organization, allowing an entity to weaken a rule means allowing all entities included to weaken the rule. The entity can be identified by its certificate;
(2) a Grid user can, at the time he runs an application, specify additional security based on the domains being deployed onto, directly in his deployment descriptor for those domains.

Finally, as active objects are active and mobile entities, there is a need to specify security at the level of such entities.

**Definition 3.** *Active Object Security*
*Security policies can be defined at the level of Active Object. Upon migration of an activity, the security policy attached to that object follows.*

In open applications, e.g. several principals interacting in a collaborative Grid application, a JVM (a process) launched by a given principal can actually host an activity executing under another principal. The principle above allows to keep specific security privileges in such case. Moreover, it can also serve as a basis to offer, in a secure manner, hosting environments for mobile agents.

**Interactions Definition** Security policies are able to control all the *interactions* that can occur when deploying and executing a multi-principals Grid application. With this goal in mind, interactions span over the creation of processes (JVM in our case), to the monitoring of activities (ActiveObjects) within processes, including of course the communications. Here is a brief description of those interactions:

- JVMCreation (JVMC): creation of a new JVM process
- NodeCreation (NC): creation of a new Node within a JVM (as the result of Virtual Node mapping)
- CodeLoading (CL): loading of bytecode within a JVM
- ActiveObjectCreation (AOC): creation of a new activity (active object) within a Node
- ActiveObjectMigration (AOM): migration of an existing activity object to a Node
- Request (Q), Reply (P): communications, method calls and replies to method calls
- Listing (L): list the content of an entity; for Domain/Node provides the list of Node/Active Objects, for Active Object allows to monitor its activity.

One must be able to express policies in a rather declarative manner. The general syntax to provide security rules, to be placed within security policy files attached to applications (for instance, see the `security` tag within the deployment descriptor), is the following:

```
Entity[Subject] -> Entity [Subject]
                                : Interaction # [SecurityAttributes]
```

Being in a PKI infrastructure, the subject is a certificate, or credential. Other "elements" (Domain, Virtual Node, Object) are rather specific to Grid applications and, in some cases, to the object-oriented framework. An "entity" is an element on which one can define a security policy. "Interaction" is a list of actions that will be impacted by the rule. Finally, security attributes specify how, if authorized, those interactions have to be achieved.

In order to provide a flavor of the system, we consider the following example.

```
Domain[inria.fr] -> Domain[ll.cnrs.fr] : Q,P # [+A,+I,?C]
```

The rule specifies that between the domain *inria.fr* (identified by a specific certificate) and the parallel machine *ll.cnrs.fr*, all communications (reQuests, and rePlies) are authorized, they are done with *authentication* and *integrity*, *confidentiality* being accepted but not required.

**Security Negotiation** As a Grid operates in decentralized mode, without a central administrator controlling the correctness of all security policies, these policies must be *combined*, *checked*, and *negotiated* dynamically.

During execution, each activity (Active Object) is always included in a *Node* (due to the Virtual Node mapping) and at least in one *Domain*, the one used to launch a JVM ($D_0$). Figure 10 hierarchically represents the security rules that

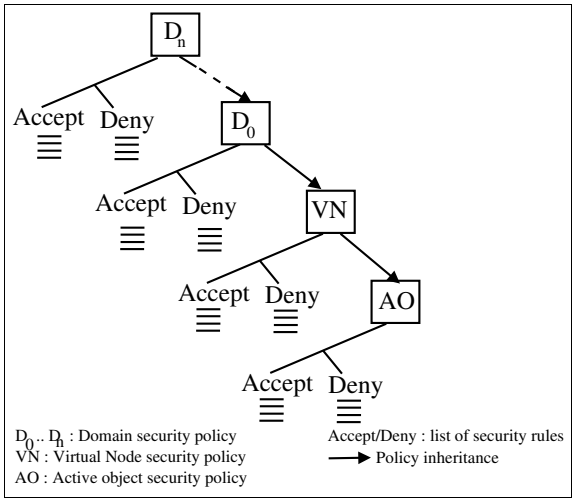



**Fig. 10.** Hierarchical security levels

can be activated at execution: from the top, hierarchical domains ($D_n$ to $D_0$), the virtual node policy (VN), and the Active Object (AO) policy. Of course, such policies can be inconsistent, and there must be clear principles to combine the various sets of rules.

There are three main principles: (1) choosing the *most specific rules* within a given domain (as a single Grid actor is responsible for it), (2) an interaction is valid only if all levels accept it (absence of weakening of authorizations), (3) the security attributes retained are the most constrained based on a partial order (absence of weakening of security). Before starting an interaction, a *negotiation* occurs between the two entities involved.

In large scale Grid applications, migration of activities is an important issue. The migration of Active Objects must not weaken the security policy being applied. When an active object migrates to a new location, three cases may happen :

- the object migrates to a node belonging to the same virtual node and included inside the same domain. In this case, all already negotiated sessions remain valid.
- the object migrates to a known node (created during the deployment step) but which belongs to another virtual node. In this case, all already negotiated sessions can be invalid. This kind of migration imposes re-establishing the object policy, and upon a change, re-negotiating with interacting entities.
- The object migrates to an unknown node (not known at the deployment step). In this case, the object migrates with a copy of the application security policy. When a secured interaction will take place, the security system

retrieves not only the object's application policy but also policies rules attached to the node on which the object is to compute the policy.

## 5 Conclusion and Perspectives

In summary, the essence of our proposition, presented in this paper, is as follows: a distributed object oriented programming model, smoothly extended to get a component based programming model (in the form of a 100% Java library); moreover this model is "grid-aware" in the sense that it incorporates from the very beginning adequate mechanisms in order to further help in the deployment and runtime phases on all possible kind of infrastructures, notably secure grid systems. This programming framework is intended to be used for large scale grid applications. For instance, we have succeeded to apply it for a numerical simulation of electromagnetic waves propagation, a non embarrassingly parallel application [21], featuring visualization and monitoring capabilities for the user. To date, this simulation has successfully been deployed on various infrastructures, ranging from interconnected clusters, to an intranet grid composed of approximatively 300 desktop machines. Performances compete with a previous existing version of the application, written in Fortran MPI. The proposed object-oriented approach is more generic and features reusability (the component-oriented version is under development, which may further add dynamicity to the application), and the deployment is very flexible.

We are conducting further works in several but complementary directions that are needed in grid computing, mainly:

- checkpointing and message logging techniques are in the way of being incorporated into the *ProActive* library. Indeed, we will as such be able to react to versatility of machines and network connections, without having to restart all the application components. Several similar works are under way ([22] for instance). The original difficulty we are faced with, is that it is possible to checkpoint the state of an active object only at specific points: only between the service of two requests (for the same reason which explains why the migration of active objects is weak). Nevertheless, we provide an hybrid protocol combining communication induced checkpointing and message logging techniques, which is adapted to the non-preemptibility of processes. This protocol ensures strong consistency of recovery lines, and enables a fully asynchronous recovery of the distributed system after a failure.
- *ProActive* components that wrap legacy codes, and in particular, parallel (MPI) native codes, are being defined and implemented (see [23] for related approaches). Of course, the design aims at enabling such components to interact with 100% *ProActive* components.

Overall, we target numerical code coupling, combination of numerical simulations and visualization, collaborative environments in dedicated application domains (see [24]), etc. The aim is to use our grid component model as a software bus for interactions between some or all of the grid components.

Indeed, the presented approach does not specifically target high-level tools appropriate for scientists or engineers who may not have a computer science background. In this respect, our objective is to succeed to incorporate the *IC2D* tools suite into grid portals, such as the Alliance portal [25]. An other complementary on going work for this class of users is to enable *ProActive* components to be published as web services (and enable those web service enabled components interact using SOAP). Notice that it does not prevent a service to be implemented as one or several hierarchical ProActive components, i.e. as the result of a recursive composition of 100% *ProActive* components, internally interacting only through *ProActive*. Then, within such portals, end-users could rely on workflow languages such as WSFL or BPEL4WS to compose applications by simply integrating some of the published components at a *coarse-grain level*.

In this way, those coarse-grain web service enabled components could provide the usual service-oriented view most users are familiar with. But as those instanciated components may use stateful resources, encompass possibly complex compositions of activities and data, we claim that the object and component oriented programming model we propose is adequate to 'internally' program and deploy those hierarchical components.

## References


1. Baude, F., Caromel, D., Huet, F., Mestre, L., Vayssière, J.: Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In: 11th IEEE International Symposium on High Performance Distributed Computing. (2002) 93–102
2. Attali, I., Caromel, D., Contes, A.: Hierarchical and declarative security for grid applications. In: 10th International Conference On High Performance Computing, HIPC. Volume 2913., LNCS (2003) 363–372
3. Baduel, L., Baude, F., Caromel, D.: Efficient, flexible, and typed group communications in java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, ACM Press (2002) 28–36
4. Baude, F., Caromel, D., Morel, M.: From distributed objects to hierarchical grid components. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003. Volume 2888., LNCS (2003) 1226–1242
5. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02). (2002)
6. Bruneton, E., Coupaye, T., Stefani, J.B.: Fractal web site. `http://fractal.objectweb.org` (2003)
7. Gannon, D., Bramley, R., Fox, G., Smallen, S., i, A.R., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., indaraju, M.G., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N.: Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. Cluster Computing **5** (2002)

8. Fox, G., Pierce, M., Gannon, D., Thomas, M.: Overview of Grid Computing Environments. Global Grid Forum document, http://forge.gridforum.org/projects/ggf-editor/document/GFD-I.9/en/1 (2003)
9. Grimshaw, A., Wulf, W.: The Legion Vision of a World-wide Virtual Computer. Communications of the ACM **40** (1997)
10. Humphrey, M.: From Legion to Legion-G to OGSI.NET: Object-based Computing for Grids. In: NSF Next Generation Software Workshop at the 17th International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, IEEE Computer Society (2003)
11. Bramley, R., Chin, K., Gannon, D., Govindaraju, M., Mukhi, N., Temko, B., Yochuri, M.: A Component-Based Services Architecture for Building Distributed Applications. In: 9th IEEE International Symposium on High Performance Distributed Computing. (2000)
12. Denis, A., Pérez, C., Priol, T.: Achieving portable and efficient parallel corba objects. Concurrency and Computation: Practice and Experience **15** (2003) 891–909
13. Denis, A., Prez, C., Priol, T., Ribes, A.: Padico: A component-based software infrastructure for grid computing. In: 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS2003). (2003)
14. Caromel, D., Belloncle, F., Roudier, Y.: The C++// Language. In: Parallel Programming using C++. MIT Press (1996) 257–296 ISBN 0-262-73118-5.
15. Baude, F., Caromel, D., Sagnol, D.: Distributed objects for parallel numerical applications. Mathematical Modelling and Numerical Analysis Modelisation, special issue on Programming tools for Numerical Analysis, EDP Sciences, SMAI **36** (2002) 837–861
16. Bull, J., Smith, L., Pottage, L., Freeman, R.: Benchmarking Java against C and Fortran for Scientific Applications. In: Joint ACM Java Grande - ISCOPE 2001 Conference, Palo Alto, CA, ACM Press (2001)
17. Caromel, D.: Towards a Method of Object-Oriented Concurrent Programming. Communications of the ACM **36** (1993) 90–102
18. Maisonneuve, J., Shapiro, M., Collet, P.: Implementing references as chains of links. In: 3d Int. Workshop on Object-Orientation in Operating Systems. (1992)
19. Dincer, K.: Ubiquitous message passing interface implementation in Java: JMPI. In: Proc. 13th Int. Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing, IEEE (1999)
20. OASIS: ProActive web site, http://www.inria.fr/oasis/ProActive/. (2004)
21. Baduel, L., Baude, F., Caromel, D., Delbe, C., Gama, N., Kasmi, S.E., Lanteri, S.: A parallel object-oriented application for 3d electromagnetism. In: IEEE International Symposium on Parallel and Distributed Computing, IPDPS. (2004)
22. Bouteiller, A., Cappello, F., Herault, T., G.Krawezik, Marinier, P.L., Magniette, F.: A fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In: ACM/IEEE International Conference on Supercomputing SC 2003. (2003)
23. Li, M., Rana, O., Walker, D.: Wrapping MPI-based Legacy Codes as Java/CORBA components. Future Generation Computer Systems **18** (2001) 213–223
24. Shields, M., Rana, O., Walker, D.: A Collaborative Code Development Environment for Computational Electro-Magnetics. In: IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software, Kluwer Academic Publishers (2001) 119–141
25. : The Alliance Portal. http://www.extreme.indiana.edu/xportlets/project/index.shtml (2004)

### 2.2.3 Object-Oriented SPMD.

L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *CCGrid 2005 : IEEE/ACM International Symposium on Cluster Computing and the Grid*, Pages 824–831, Vol. 2. April 2005.

2005 IEEE International Symposium on Cluster Computing and the Grid

# Object-Oriented SPMD

Laurent Baduel, Françoise Baude, Denis Caromel
INRIA - CNRS - University of Nice Sophia-Antipolis
2004, Route des Lucioles - BP93 - 06902 Sophia Antipolis Cedex France
Email: First.Last@sophia.inria.fr Tel: +33 4 92 38 75 56 Fax: +33 4 92 38 76 44

*Abstract*— **This article presents an evolution of classical SPMD programming for clusters and grids.**

**It is named "Object-Oriented SPMD" as it is based on remote method invocation. More precisely, it is based on an active object pattern, extended as a typed group of active objects, to which SPMD's specificities are added. The proposed programming model is more flexible: techniques to postpone barrier and to remove any explicit loop make it possible to privilege reactivity and reuse.**

**The resulting OO-SPMD API has been implemented in *ProActive*. Good scalability and quite competitive performances, compared to what is obtained using C-MPI, are demonstrated.**

**Keywords**: Grid Computing, SPMD Programming, Java, *ProActive*.

## I. Introduction: Context and Related works

### A. SPMD programming

SPMD stands for Single Program Multiple Data. SPMD programming is a common way to organize a parallel program, on both clusters of workstations and parallel machines, and more recently also on grids [1]. A single program is written and loaded onto each node of a parallel computer. Each copy of the program runs independently, coordination events apart. So the instruction streams executed on each node can be completely different, alas for the most common pattern, i.e., master-slave, only two different streams are needed. Each copy of program (process) owns a rank number: a unique ID. The specific path through the code is in part selected by this ID.

Traditionally, in the SPMD model, the language itself does not provide implicit data transmission semantics. In general, the communication patterns are **explicit message-passing** implemented as library primitives. This simplifies the task of the compiler, and encourage programmers to use algorithms that exploit locality. Data on remote processors are accessed exclusively through explicit library calls.

SPMD model maps easily and efficiently to distributed and to parallel applications and distributed memory computing. The most famous environments implementing a message-passing SPMD model are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

### B. SPMD programming with an Object-Oriented flavour

*1) Message-Passing SPMD:* In the 1990's, due to the increasing success of object-oriented programming, many research groups have experimented the idea to both combine the usage of an object-oriented programming language (such as C++ or Java) and MPI (or PVM) for writing and running parallel and distributed applications. One of the precursors has been the MPI-2 specification itself, collecting the notions of the MPI standard as suitable class hierarchies in C++, and defining most of the library functions as class member functions. This specification has been further extended in Object-Oriented MPI [2] in order to be able to deal with the transmission of objects. Essentially, OOMPI provides mechanisms to build user-defined data types according to the MPI spec, in order to represent those objects, and further communicating them. Those approaches have been even further developed with the success of Java and have eventually lead to two main categories of propositions for having message-passing SPMD within Java:

- a wrapping of the native MPI implementation library itself within the object oriented language (e.g. mpiJava [3], or JavaMPI [4] where wrappers are automatically generated)
- an *MPI-like* implementation of a message-passing specification as MPI, written using the object-oriented language itself, and available as a library. Notably, MPIJ [5] which seeks to be competitive with native MPI implementations. The most achieved is MPJ [6], in which notions such as Communicators, Datatype for the type of the elements in the message buffers, etc, are modeled as classes.

Overall, in the early 2000's, those works – done under the auspices of the JavaGrande Forum [7] – were considered as a first phase in a broader venture to define a more Java-centric high performance message-passing environment. The main aim was to succeed to conciliate both performance and portability, while not departing from the consensual goal of offering MPI-like services to Java programs.

*2) Remote method based SPMD:* All propositions grounding up on remote method invocation for communication among activities take for granted that this enables the exchange of any typed data, by automatic marshaling-unmarshaling. Clearly, this better suits to the object oriented paradigm than explicit message-passing, in which send and receive must be explicitly programmed in matched pairs. One work grounding on Java remote method invocation, but generalizing it so it can supports communication between more than two parties is CCJ [8]. Specifically, CCJ aims at adding collective operations to Java's object model (implementing everything on top of RMI). Parallel activities are expressed as threads groups and not as objects groups (in fact, activities in Java are expressed by threads which are orthogonal to objects). As threads may belong to several groups, this implies that any method of the CCJ API (e.g. `barrier, broadcast, reduce,...`) aiming at executing an MPI-like collective operation must have the reference of the group of threads as parameter (in

0-7803-9074-1/05/$20.00 ©2005 IEEE

a similar way as passing the communicator as parameter in any MPI communication). Also, in CCJ, all threads have the same program and, in particular, any collective operation must be called by all threads in the implied group. Differently to the approach followed in CCJ, another concept for collective communications is to group Java objects into *groups*, and extend the remote method invocation mechanism such that it transparently applies to a group of possibly remote objects. It fits much better in the object-oriented approach: triggering the execution of a chunk of code (described in any public method in the class) in parallel is done simply by calling the corresponding method on the group, remotely and possibly asynchronously. By doing this, remote method invocation is exploited as the *only* communication mechanism between any number of remote activities.

Having a group of objects towards which methods are invoked is usually considered to be a suitable OO abstraction for building *distributed* applications – even if it usually requires the additional usage of multicast delivery protocols such as causally or totally order delivery. The suitability of groups and associated group method invocation mechanisms are more rarely studied as a suitable support for parallel computing (notable exceptions being GMI [9] in Java, ARMI [10] in C++).

### C. Contribution

In this paper, we propose a pure object-oriented SPMD programming model as an extension of a typed group communication mechanism we previously defined in [11]. For this, the objects groups supporting the distributed computation will also be further organized following a topology, i.e. adding the notion of an ID for each member in the SPMD group and the way to easily reference its neighbors. Collective operations will be revisited and extended with barrier synchronization such as providing a complete *Object Oriented SPMD* model.

The solution we propose is grounded on *ProActive*, a strongly proven programming [12] and deployment model for distributed object-based computations, on any distributed memory platforms including grids [13], [14]. *ProActive* is based on the active object paradigm, and moreover featuring a well defined semantics of the computing model [15]. In this respect, the SPMD programming solution we define is a smooth and perfectly integrated extension of the active object principle. We want to demonstrate to the programmer that using it, he can define programs grounded on a single concept, the *active object*. Using this paradigm, he can seamlessly target the whole spectrum of applications: from sequential mono-threaded, concurrent and multi-threaded, distributed, up to parallel and distributed ones.

To our knowledge, a proposition which is close to ours is GMI [9], (in the objective and in the way to achieve it). A strong difference comes from the fact that GMI generalizes Java RMI. As such, it is confronted with its constraints, specially, the need for the programmer to take care of possible concurrent executions of a same method (implying to mix functional code with the usage of regular Java monitor mechanisms). On the contrary, the active object pattern is a cleaner abstraction for distributed computing, and as such should ends up easier for programing Object-Oriented SPMD applications.

Section II presents briefly the *ProActive* library. Section III presents the typed group communication of *ProActive* and the recent optimizations we have added to it. Section IV introduces the complete Object-Oriented SPMD programming model. One strong advantage is that the corresponding API is very light: only primitives for SPMD group membership and barrier synchronizations are required. Indeed, all point-to-point and collective communications are implicit as the focus is more on which method to execute in parallel instead of how to effectively manage the parallel and distributed associated aspects. Section V presents benchmarks on large configurations, including comparisons with MPI. Section VI concludes.

## II. The *ProActive* library

*ProActive* is an LGPL Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, *ProActive* provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on clusters, or on grids.

As *ProActive* is built on top of the Java standard API[1], it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

*a) Base model:* A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [12]. Contrary to classical RMI, all kinds of method call parameters towards an active object are passed by (deep-)copy. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee. The *ProActive* library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

*b) Mapping active objects to JVMs: Nodes:* Another extra service provided by *ProActive* (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs, and to add a few services. *Nodes* provide those extra capabilities: a *Node* is an object defined in *ProActive* whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional

[1] mainly Java RMI and the Reflection API

way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance `rmi://lo.inria.fr/node`.

Let us take a standard Java class `A`. The instruction:

```
A a = (A) ProActive.newActive("A",params, N1);
```

creates a new active object of type `A` on the JVM identified with `N1`, for instance `rmi://lo.inria.fr/node`. Further, all calls to that remote object will be asynchronous, and subject to the *wait-by-necessity*:

```
a.foo (...);          // Asynchronous call
v = a.bar (...);      // Asynchronous call
...
v.f (...);          // Wait-by-necessity:
          // wait until v gets its value
```

Compared to traditional futures, *wait-by-necessity* offers two important features: (1) futures are created implicitly and systematically, (2) futures can be passed to other remote processes.

Note that an active object can also be bound dynamically to a node as the result of a migration. In order to help in the deployment phase of *ProActive* components, the concept of virtual nodes as entities for mapping active objects has been introduced [13]. Those virtual nodes are described externally through XML-based deployment descriptors which are then read by the runtime when needed. The goal is to be able to deploy an application anywhere without having to change the source code, all the necessary information being stored in those descriptors. As such, deployment descriptors provide a mean to abstract from the source code of the application any reference to software or hardware configuration. It also provides an integrated mechanism to specify external processes (e.g. JVM) that must be launched, and the way to do it.

## III. Typed Group Communications

The group communication mechanism of *ProActive* efficiently achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies.

### A. Summary of group communication

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remains *typed*. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.

On the standard Java class `A` used above, here is an example of a typical group creation:

```
  // A group of type "A" and its 3 members
  // are created at once on the nodes
  // directly specified, parameters are
  // specified in params,
Object[][] params = {{...}, {...}, {...}};
A ag = (A) ProActiveGroup.newGroup("A",
             params, {node1,node2,node3});
```

Elements can be included into a typed group only if their class equals or extends the class specified at the group creation. Note that we allow and handle *polymorphic* groups. For example, an object of class `B` (`B` extending `A`) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group. Groups can also be dynamically modified, adding or removing members, getting a group from a typed group.

A method invocation on a group has a syntax similar to a standard method invocation:

```
ag.foo(...); // A group communication
```

Such a call is propagated to all members of the group using multithreading: a method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be asynchronous, and if the member is a standard Java object, the method call will be a standard Java method call (within the same JVM). By default, the parameters of the invoked method are broadcasted to all the members of the group.

An important specificity of the group mechanism is: the *result* of a typed group communication *can also be a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results, as shown in Figure 1: *"result group"*. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited then the caller blocks, but as soon as one reply arrives in the result group the method call on this result is executed. E.g. in

```
   // A method call on a group with result
V vg = ag.bar();
   // vg is a typed group of "V"
   // This is also a collective operation:
vg.f();
```

a new `f()` method call is automatically triggered as soon as a reply from the call `ag.bar()` comes back in the group `vg` (dynamically formed). The instruction `vg.f()` completes when `f()` has been called on all members: this constitutes a local synchronization point from the point of view of the initiator of the group method call, i.e., certifying that all peers in the group `ag` have executed the method `bar()`. Another remark is that collected results, and thus *gathered* through the `vg` group can subsequently be merged. This is like achieving a global reduction. The reduction operator can be any user defined method (such as `f()` in the above example), and moreover, the operator can be applied as soon as each result comes back. So, even if the reduction operation is not executed in parallel, its cost can be hidden by the transmission of the not yet arrived results.

Other features are available regarding group communications: parameter dispatching using groups (through the definition of *scatter groups*), hierarchical groups, dynamic group manipulation (`add`, `remove` of members), explicit group synchronization (`waitOne`, `waitAll`, `waitAndGet`); see [11] for further details and implementation techniques.

### B. Optimizations

The group communication mechanism is built upon the *ProActive* elementary mechanism for asynchronous remote method invocation with automatic future for collecting a reply. As this last mechanism is implemented using standard

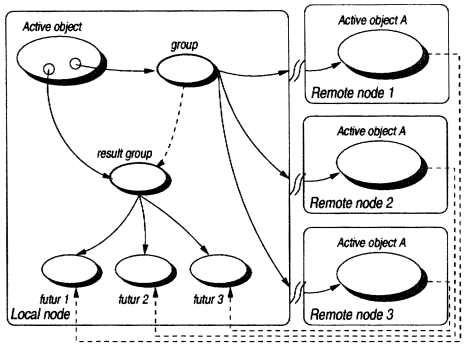



Fig. 1. Method call on group

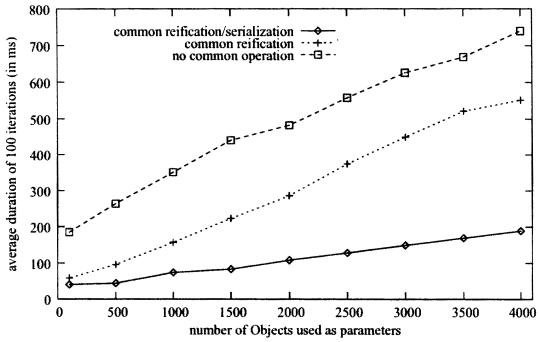



Fig. 2. Factorization of common operations

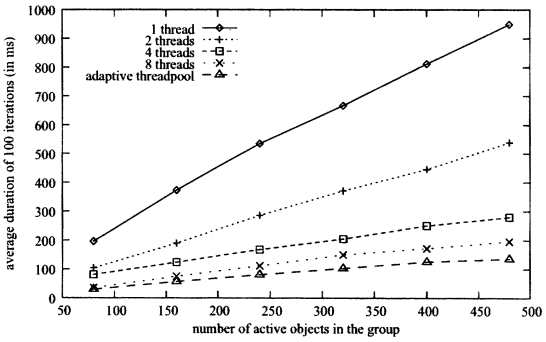



Fig. 3. Adaptative threadpool

Java, such as RMI, the group mechanism is itself platform independent. A group communication must be thought as a replication of more than one (say N) *ProActive* remote method invocations towards N objects. Of course, we incorporated optimizations into the group mechanism implementation.

*a) Common operations factorization:* Many operations are common while invoking a method on a group of objects. Those operations may be factorized. First is the reification operation that transforms the method invocation into a Java object using the Meta Object Protocol. This operation involves reflection techniques that are known to be expensive. The method being the same for all group members, the operation is done just once.

Second point subject to factorization is the serialization of the method parameters sent during the group communication. As the serialization process is very slow, we want to avoid the repetition of this operation. Before the RMI mechanism steps in, the parameters (and codebase informations) are converted into a byte array to be more efficiently sent several times by RMI. This does not apply in the case of scatter group in which parameters for each member differ.

Figure 2 presents the average time (in milliseconds) spent to perform one hundred method invocations depending on the amount of data to send (objects used as parameters). The group contains 80 objects distributed on 16 machines (cluster of PIII @ 933MHz interconnected with a 100Mb/s ethernet network). The upper curve exposes the performances without any operation factorized. The curve in the middle plots the performances obtained by factorizing the reification operations. The last curve represents the performances obtained by factorizing the reification operations and the serialization. This allows better performances (up to a 3.9 ratio in the Figure 2).

*b) Adaptive threadpool:* Using several threads allows to send messages simultaneously. Doing this way, the delays required by RMI to make the rendez-vous with each remote object are recovered and no more added. In order to maintain the *ProActive* method invocation semantic, we introduce a synchronization. We extend the notion of rendez-vous for group communication: doing this, an asynchronous group communication blocks until the method invocation has reached all group members.

Because group membership is dynamic, a fixed number of threads used to communicate with the group members is not appropriate. Likewise, a one-to-one group member / thread ratio is not suitable: too many threads will harm performances. Our solution is to associate to a group an adaptive pool of threads. The member / thread ratio may be defined by the programmer depending of the requirement of its application (default value is 8).

Figure 3 plots the average time (in milliseconds) spent to perform one hundred method invocations depending on the number of objects in a group. The group members are distributed on 16 machines (cluster of PIII @ 933MHz interconnected with a 100Mb/s ethernet network). The curves represent the performances depending on the number of threads used to make the calls. The more we used threads the smaller is the delay to make the group communication. The four upper curves are associated with a fixed number of threads. The lower one is associated with a dynamic number of threads. It shows better performance, because the number of threads is (automatically and transparently) at any moment the adequate number needed.

## IV. Object-Oriented SPMD

The proposed active objects group mechanism presented in section III is already a usable and even efficient basis to program non embarrassingly parallel applications using a pure object-oriented paradigm, i.e. using only object-oriented method invocation for e.g. computational electromagnetism [14], [16]. But, some of the features specific to SPMD programing were lacking, and their addition constitutes the core

of this section. We name the resulting proposition as *Object Oriented SPMD* (OO-SPMD for short).

### A. Requirements

These specificities fall into three categories:

- identification of each member taking part in the parallel computation, and concept of member position relatively to the others (i.e. neighboring relation among members)
- expression of the program run by each member taking part in the parallel computation. In pure object groups based paradigms (e.g. as GridRPC for grid computing on Network Enabled Servers like NetSolve [17] or Ninf [18]), members act in a sense as *passive* servers only activated by method calls triggered by clients. Servers do not have their own activity. On the contrary, in SPMD computing, all members are active by their own even if, for simplicity, they all execute the same program (e.g., in all flavors of MPI, in CCJ [8], in GMI [9], this program is run by the main thread on each process or participating JVM). In *ProActive*, each active object is by essence the support of a proper activity (there is no *main*, but a *runActivity* method). This activity aims at enacting the sequential service of requests (see paragraph II.a). So, in our approach, the SPMD program will not be expressed as a classical *big loop*, but as the implicit result of a succession of request services executed in FIFO order. As will be emphasized below, this way of expressing the core of any member's SPMD program enables behaviors pertaining to reactivity, evolutivity, dynamicity usually considered to be far away from the traditional SPMD model.
- full range of collective operations (communication and global synchronization) among the members. Considering the presentation of the typed group communications in section III, only the expression of global synchronization barriers is lacking and so needs to be considered below.

### B. Main principles of OO-SPMD

An OO-SPMD group is defined as follows: it is a group of active objects where each member has a reference, a group proxy, towards the group itself (see Figure 4). Each active object in the SPMD group is also provided with a specific *rank* in the group.

```
  // A group of type "A" and its members
  // are created at once by an external
  // active object
Object[][] params = {{...}, {...}};
A ag = (A) ProSPMD.newSPMDGroup("A",
                  params, {Node1,...});
  // The computation on each member may
  // now be started, i.e. invoking a method
  // called e.g compute() defined in class A
ag.compute();
```

On each group member created, one of the first actions to run is to get the reference of the group it belongs to, the rank, etc. One must be careful to clearly distinguish a classical Java reference to the object (this), and a *ProActive* asynchronous reference to it, as an active object. This last one enables the active object to implement the parallel task. Traditionally in SPMD, the parallel task is expressed as an iterative or recursive loop, which essentially handles message receptions and triggers the corresponding treatment, according to the message's tag (a `case` or a `if` control structure is usually programmed). In OO-SPMD, the parallel task on any member of the SPMD group is run by repeatitively invoking asynchronous methods to itself (so, the need to have an asynchronous reference). A member triggers data receptions and the corresponding treatment through the asynchronous service of methods remotely called by other members in the group. All method services are FIFO-ordered.

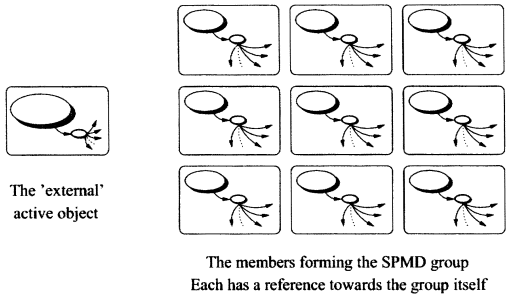



Fig. 4. An SPMD group

```
  // A reference to the typed group I belong to
A a = (A) ProSPMD.getSPMDGroup();
  // An asynchronous reference to myself
A me = (A) ProActive.getStubOnThis();
  // My rank in the group
int rank = ProSPMD.getMyRank();
 // Start the 'iterative' loop by sending
 // myself an asynchronous method call
me.loop();
 // To iterate, loop() again calls me.loop()
```

Moreover, in a traditional SPMD program, execution control is exclusively based on `if` statements and process ID or rank numbers. In our approach, switching execution control can be also based on dynamically created groups at any moment at runtime. Such groups can be derived from existing ones (subgroups, or group combination for instance) or according to any kind of properties (rank, fields of the object, ...).

### C. Topologies

To simplify the access to neighbors in the group with which a given member must communicate according to the parallel algorithm, it is useful if the SPMD group is further organized according to Cartesian topologies (as in MPI). At this time, we offer the following: line, plan, ring, cube, hypercube, torus, torusCube (torus in 3 dimensions) but, contrary to statically designed topologies, the addition of new topologies is open. Topologies may also be obtained from an other topology. Here is an example:

```
  // Organize my group as a 2D plan
Plan topology = new Plan(a, WIDTH, HEIGHT);
  // Get a ref. to my neighbors in the plan
A left = (A) topology.left(me);
```

```
A down = (A) topology.down(me);
...
  // One-way communication with neighbors
  // in an asynchronous fashion
left.foo(params);
down.foo(params);
...
  // Get a ref. to the topology formed by
  // the first line of the plan
Line line = topology.line(0);
```

The notion of neighborhood is strongly attached to the topology. By extending a topology, the programmer may redefine the neighborhood to best fit the needs of the application.

### D. Synchronization barriers

The only collective behavior related methods of our OO-SPMD API pertain to global barriers. Indeed, as already explained in section III, all collective (resp. point-to-point) communications within the group can be expressed as applicative-level method calls triggered via the group proxy (resp. via the asynchronous reference of the target member).

The standard definition of a global barrier is that all members in the group (or those enrolled in the barrier, see below) must not proceed further in their computation while not all the members have reached the barrier. Given the active object model, we propose a slightly different but more appropriate semantic: from the viewpoint of a member reaching a barrier, it is effective (i.e. it blocks the member) only in the future: more precisely the exact moment when the current service has terminated. In practical terms, all instructions lying after the barrier in the current method being served will be executed, so care must be taken (see an example in subsection V-B). Nevertheless, the meaning of what is a global synchronization barrier is as usual, but instead of pertaining to the next instruction, it pertains to the next request's service: when encountering a barrier, the service of the first request waiting in the request queue will be able to proceed on any enrolled member only when all have reached the barrier.

Technically, when an active object executes a call to a global barrier this triggers the storage in the front of its request queue of a specific token. Associated to this token is the total number of members (including the member itself) to wait for, i.e. that must reach the barrier. Each time a given global barrier is reached by a member, this triggers the decrementation of this number on each member enrolled in the barrier. Eventually, the barrier is released on each enrolled member, as soon as the number reaches zero.

Actually, we propose three kinds of barriers, two globals and one more local:

- A *total barrier*, within which a string parameter represents a unique identity name for the barrier. It is assumed that this blocks all the members in the SPMD group.
  `ProSPMD.barrier("MyBarrier");`
- A *neighbor barrier*, involving not all the members of an SPMD group, but only the active objects specified in a given group. Those objects, that contribute to the end of the barrier state, are called neighbors as they are usually local to a given topology. An active object that invokes the neighbor barrier must be in the group given as parameter.
  `ProSPMD.barrier("Bar", neighborsGroup);`
- A *method barrier* stops the active object that calls it, waiting for a request on all the specified methods to be served. The order of the methods does not matter, nor the active objects they come from. As such, this barrier is purely local, and does not trigger extra messages to be exchanged as the two others.
  `ProSPMD.barrier({"foo","bar","gee"});`

## V. Example and Benchmarks

We illustrate OO-SPMD with a concrete example. We choose *Jacobi iterations* because it is a simple application, easy to distribute in a traditional SPMD manner. The algorithm performs local computation and communication to exchange data. The Jacobi method is a method of solving a linear matrix equation. Each element is solved by computing the mean value of the adjacent values. The process is then iterated until it converges; it means until the difference between old and new value in absolute becomes lower than a given threshold.

The following code shows the main loop (an iteration based loop) of a solver. At each iteration, the value at a point is replaced by the average of the up, down, left, and right neighbor values. External boundary values are fixed statically at the beginning of the application and do not change at runtime.

```
while (!converged) {
  for (y=1 ; y<MATRIX_HEIGHT-1 ; y++) {
    for (x=1 ; x<MATRIX_WIDTH-1 ; x++) {
      new(x,y) = ( old(x,y-1) + old(x,y+1) +
                   old(x-1,y) + old(x+1,y) )/4;
      if (abs(new(x,y)-old(x,y)) < THRESHOLD) {
        converged = true;
      }
      exchange(new,old);
} } }
```

The structure of this code is quite simple, so we use a coarse-grained data-parallel approach to transform it into a similar parallel code. The arrays `old` and `new` are distributed over nodes taking the form of active objects. Each active object, named `SubMatrix`, is responsible for receiving boundary values from adjacent sub-matrixes and computing its own part of data.

The parallel algorithm depends of the data distribution scheme. We choose a two-dimensional distribution scheme. As shown in figure 5, communications occur at block boundaries. So the amount of data exchanged is minimized by the two-dimensional distribution which has a better internal area / border ratio. With this partition, each sub-matrix may communicate with two, three, or four neighbors, depending of their position (respectively at a corner, a border, or in the center of the whole matrix). This partition is more effective when the data to processor ratio is large.

Communications appear at sub-matrix boundaries to send boundaries values to neighbors and receive values of neighbors. A copy of the boundary of each sub-matrix is present in its neighbor sub-matrix. Storage of boundary data is allocated at the producer, and at the consumer sub-matrixes. This is a static allocation because the size and the location of boundary

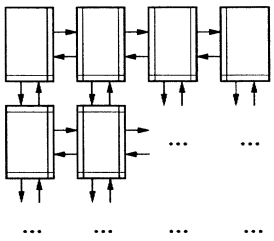


Fig. 5. Distributed algorithm

buffers is fixed and never evolve during Jacobi. This is induced by the Jacobi algorithm itself, but if needed, our framework could support strongly dynamic algorithms.

### A. MPI Jacobi

Using a message passing approach based on asynchronous send and receive, with the MPI library, the resulting parallel code is something like:

```
while (!converged) {
  internal_compute(&converged);
  MPI_Send(north_border, SUBMATRIX_WIDTH,
           MPI_DOUBLE, north, 1,
           MPI_COMM_WORLD, &status);
  MPI_Recv(border_received_from_north,
           SUBMATRIX_WIDTH, MPI_DOUBLE,
           north, 1, MPI_COMM_WORLD, &status);
    // send and receive for south, east, west
  ...
  boundaries_compute(&converged);
  exchange(new,old);
} } }
```

The send and receive operations are repeated for each communication with a neighbor (up to 4), even if the operations are the same.

### B. OO-SPMD Jacobi

Using our OO-SPMD approach, the code becomes much concise. The whole matrix is distributed and understood as a two-dimensional topology using the `Plan` topology. The neighborhood of any `SubMatrix`, named `neighbors` in the example, is automatically obtained through methods of the `Plan` topology.

```
me = ProActive.getStubOnThis();
public void jacobiIteration() {
  internal_compute(); //updates converged
  neighbors.send(boundariesGroup);
  ProSPMD.barrier({"send", ... ,"send"});
  me.boundaries_compute(); //updates converged
  me.exchange();
  if (!converged) me.jacobiIteration();
}
```

Synchronization is done by data flow, and the barrier ensures that the sub-matrix and its neighbors have exchanged their own boundaries values before computing the whole boundaries. The method calls performed after the barrier must be asynchronous (put in the queue of the active object), otherwise they would be served immediately, i.e. before the execution of the barrier. Overall, according to the semantic of the *method barrier*, the data (i.e. parameter of `send`) will have been exchanged before the barrier will be released, guarantying that any member gets the data in order to compute the boundaries values (`boundaries_compute`).

Data communications to all neighbors is performed using a *scatter group* (the group of boundaries values `boundariesGroup`): as the real parameter of the *send* method is a group declared as of type *scatter*, it is transparently scattered to each member of the *neighbors* group. As for the MPI version, the construction of the structures containing boundaries values was not specified on this chunk of code. It only consists of building a group containing the boundaries.

A very interesting property of our model is that it remains *reactive*. It means that any part or any member of an OO-SPMD application may also serve incoming method call requests incoming from another application. This is allowed by the fact that the parallel task is expressed as asynchronous calls to a method (`jacobiIteration` for instance): an external request is thus able to come in between requests addressed to the active object. We think this flexibility is very appropriate to one of the many possible applications of grid computing: the coupling of, on one side a parallel object-oriented SPMD computation, and on the other side, an external and remote application that is in charge of, for instance, steering, visualization, etc.

### C. Benchmarks

*1) Data scalability:* The first benchmark, presented by the Figure 6, uses a cluster of 16 bi-Pentium III @ 933 Mhz 512MB (SDRAM) - 256 Kb L2 cache, Linux RedHat 2.4.20, interconnected with a 100 Mb/s Ethernet. Even if machines are bi-processor, we used only one processor per machine during our experimentations. For the C/MPI version we used gcc 3.3.2 and MPICH 1.2.5.2. For the Java version, we used the Sun Java Virtual Machine 1.5.0.

The graphic presents the average duration, in milliseconds, of one Jacobi iteration depending of the data contained on each node, in millions of double. The average time was computed after 100 iterations. Of course, the C language with MPI remains more efficient than Java with RMI. But the ratio of 3.3 (average) of performance is maintained despite the growth of data (see the bold curve). It is interesting to notice that 3.3 is also the ratio of performance between Java and C for the sequential versions. Our approach thus allows an efficient distribution and is scalable regarding data. For 29M of doubles, speedup of the C/MPI version is 15.41, speedup of the Java/OO-SPMD is 15.23.

*2) Deployment scalability:* Figure 7 presents the Jacobi application running on up to 130 machines. This experimentation was done using a Peer-to-Peer deployment scheme provided by *ProActive* within an Intranet configuration. The machines used are desktop computers, simultaneously used by their users. They are heterogeneous (slowest is a Pentium III @ 993 MHz 512MB, fastest is a bi-Pentium IV @ 3,2 GHz 2GB), they are interconnected with 100 Mb/s network, they are running under Linux (with different kernel versions). Deployed applications run with a lower priority (nice 19) in order to not disturb regular users. We used the Sun Java Virtual Machine 1.4.2.

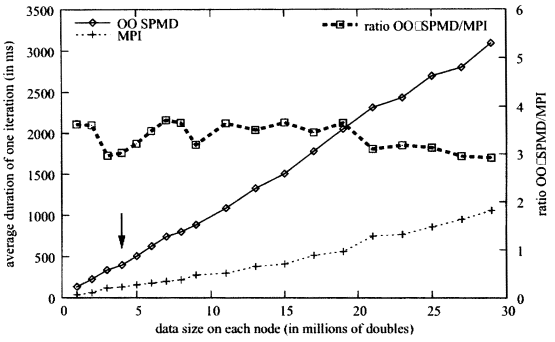



Fig. 6. Benchmark: C/MPI and Java/OO-SPMD versions

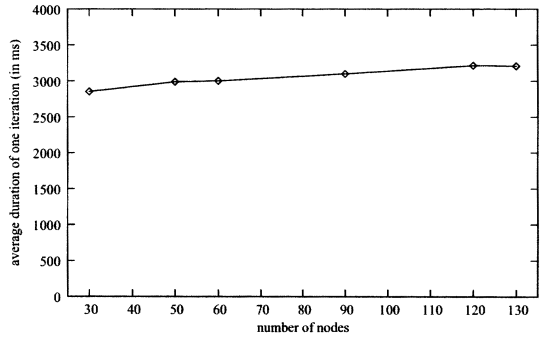



Fig. 7. OO-SPMD scalability in a peer-to-peer experiment

For all measurements, each node is responsible for the same amount of data (2000x2000 doubles). The overall size of the problem grows with the number of nodes involved in the computation. The line plots the average duration, in milliseconds, of one Jacobi iteration depending of the number of nodes involved. As previously, the average time was computed after 100 iterations. Compared to the previous benchmarks, for the same amount of data per node (see arrow in Figure 6), execution is 7 times slower. We blame the lower priority of execution and the older JVM for this loss of performance. Besides, the performance remains regular, regardless of the number of used nodes. From this, we conclude that the application is scalable.

## VI. Conclusion

We have introduced a parallel programming model, which we name *Object-Oriented SPMD* as an alternative to the traditional Message-Passing SPMD style. Overall, it allows more flexibility, and a higher level of abstraction. First, it enforces members taking part in the computation just the required involvement in collective operations. E.g. in MPI, a call to `MPI_broadcast` must be run by all members, even if for all except the sender, this call aims only at receiving the message. On the contrary, using our solution, a method invocation towards a single active object to trigger a point-to-point interaction, or towards a SPMD group of active objects to trigger a collective interaction between all the members only differ by the target object reference. This way, we promote asychronous remote method invocation and the active object pattern as the only required communication and structuration mechanism. Secondly, our approach to SPMD programming has potential for evolution. Instead of defining the parallel task as a single 'big' loop as in traditional SPMD programming, OO-SPMD enables to receive and treat data in a more flexible order (discarding the need to program sometimes intricate `case` statements depending of received message's tag).

The resulting OO-SPMD API already forms part of the *ProActive* open-source library, freely distributed through the Object Web consortium for open-source middleware. Our ambition is to have this approach used on real size applications. We already successfully applied the typed group communication mechanism to solve simulation in electromagnetism [14], [16]. Our current work is to apply the whole OO-SPMD approach to it. Next, we plan to target other application domains, such as biogenetics (applying BLAST in parallel), for which we already have developed applications, but not yet using OO-SPMD.

## References

[1] G. Fox, M. Pierce, D. Gannon, and M. Thomas, "Overview of grid computing environments," Global Grid Forum, Tech. Rep., 2002.

[2] J. Squyres, B. McCandless, and A. Lumsdaine, "Object Oriented MPI: A Class Library for the Message Passing Interface," in *POOMA'96*, http://www.osl.iu.edu/download/research/oompi/oompi.pdf.

[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "mpiJava: An Object-Oriented Java interface to MPI," in *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*.

[4] S. Mintchev and V. Getov, "Towards portable message passing in Java: Binding MPI," in *Recent Advances in PVM and MPI*, ser. LNCS, no. 1332, 1997.

[5] G. Judd, M. Clement, and Q. Snell, "DOGMA: Distributed Object Group Metacomputing Architecture," *Concurrency: Practice and Experience*, vol. 10, no. 11/13, 1998.

[6] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "MPJ: MPI-like message passing for Java," *Concurrency: Practice and Experience*, vol. 12, no. 11, pp. 1019–1038, 2000.

[7] "Java Grande Forum," www.javagrande.org.

[8] A. Nelisse, T. Kielmann, H. E. Bal, and J. Maassen, "Object-based Collective Communication in Java," in *Joint ACM Java Grande - ISCOPE Conference*. Palo Alto, California, USA: ACM Press, June 2001, pp. 11–20.

[9] J. Maassen, T. Kielmann, and H. Bal, "GMI: Flexible and Efficient Group Method Invocation for Parallel Programming," in *LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 2002.

[10] S. Saunders and L. Rauchwerger, "ARMI: An Adaptive, Platform Independent Communication Library," in *PPoPP'03*.

[11] L. Baduel, F. Baude, and D. Caromel, "Efficient, Flexible, and Typed Group Communications in Java," in *Joint ACM Java Grande - ISCOPE Conference*. Seattle: ACM Press, 2002, pp. 28–36.

[12] D. Caromel, "Towards a Method of Object-Oriented Concurrent Programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, September 1993.

[13] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière, "Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications," in *11th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, 2002, pp. 93–102.

[14] L. Baduel, F. Baude, D. Caromel, C. Delbe, N. Gama, S. E. Kasmi, and S. Lanteri, "A parallel object-oriented application for 3d electromagnetism," in *IEEE International Symposium on Parallel and Distributed Computing, IPDPS*, april 2004.

[15] D. Caromel, L. Henrio, and B. Serpette, "Asynchronous and deterministic objects," in *31st ACM Symposium on Principles of Programming Languages*, 2004.

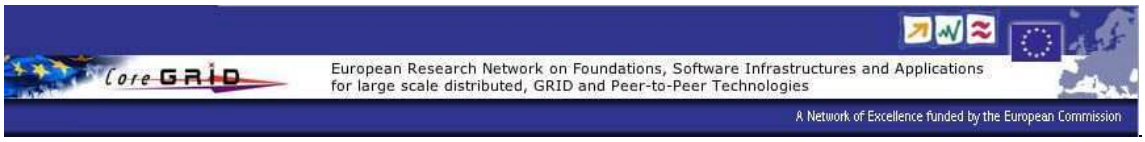
[16] F. Huet, D. Caromel, and H. E. Bal, "A High Performance Java Middleware with a Real Application," in *SuperComputing 2004*.

[17] "NetSolve," http://icl.cs.utk.edu/netsolve.

[18] "Ninf-G," http://ninf.apgrid.org/.

### 2.2.4 Grid Application Programming Environments.

F. Baude, D. Caromel, F. Huet, T. Kielmann, A. Merzky, and H.Bal. *Future Generation Grids*, chapter Grid Application Programming Environments. CoreGRID series. Springer, jan 2006. ISBN : 0-387-27935-0. Also as the CoreGrid TR003 report, `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0003.pdf`.




# Grid Application Programming Environments

*Thilo Kielmann, Andre Merzky, Henri Bal*
*Vrije Universiteit*
*Amsterdam, The Netherlands*
{kielmann,merzky,bal}@cs.vu.nl
*Francoise Baude, Denis Caromel, Fabrice Huet*
*INRIA, I3S-CNRS, UNSA*
*Sophia Antipolis France*
{fbaude,dcaromel,fhuet}@sophia.inria.fr

---


CoreGRID Technical Report
Number TR-0003
June 21, 2005

Institute on Problem Solving Environment, Tools and GRID Systems

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

# Grid Application Programming Environments

Thilo Kielmann, Andre Merzky, Henri Bal
Vrije Universiteit
Amsterdam, The Netherlands
{kielmann,merzky,bal}@cs.vu.nl
Francoise Baude, Denis Caromel, Fabrice Huet
INRIA, I3S-CNRS, UNSA
Sophia Antipolis France
{fbaude,dcaromel,fhuet}@sophia.inria.fr

*CoreGRID TR-0003*
June 21, 2005

**Abstract**

One challenge of building future grid systems is to provide suitable application programming interfaces and environments. In this chapter, we identify functional and non-functional properties for such environments. We then review three existing systems that have been co-developed by the authors with respect to the identified properties: ProActive, Ibis, and GAT. Apparently, no currently existing system is able to address all properties. However, from our systems, we can derive a generic architecture model for grid application programming environments, suitable for building future systems that will be able to address all the properties and challenges identified.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

# 1 Introduction

A grid, based on current technology, can be considered as a distributed system for which heterogeneity, wide-area distribution, security and trust requirements, failure probability, as well as high latency and low bandwidth of communication links are exacerbated. Different grid middleware systems have been built, such as the Globus toolkit [39], EGEE LCG and g-lite [12], ARC [35], Condor [20], Unicore [18], etc). All these systems provide similar grid services, and a convergence is in progress. As the GGF [22] definition of grid services tries to become compliant to Web services technologies, a planetary-scale grid system may emerge, although this is not yet the case. We thus consider a grid a federation of different heterogeneous systems, rather than a virtually homogeneous distributed system.

In order to build and program applications for such federations of systems, (and likewise application frameworks such as problem solving environments or "virtual labs"), there is a strong need for solid high-level middleware, directly interfacing application codes. Equivalently, we may call such middleware a grid programming environment. Indeed, grid applications require the middleware to provide them with access to services and resources, in some simple way. Accordingly, the middleware should implement this access in a way that hides heterogeneity, failures, and performance of the federation of resources and associated lower-level services they may offer. The challenge for the middleware is to provide applications with APIs that make applications more or less grid unaware (i.e. the grid becomes invisible).

Having several years of experience designing and building such middleware, we analyze our systems, aiming at a generalization of their APIs and architecture that will finally make them suitable for addressing the challenges and properties of future grid application programming environments. In Section 2, we identify functional and non-functional properties for future grid programming environments. In Section 3, we present our systems, ProActive, Ibis, and GAT, and investigate which of the properties they meet already. In Section 4, we discuss related systems by other authors. Section 5 then derives a generalized architecture for future grid programming environments. In Section 6 we draw our conclusions and outline directions of future work.

# 2 Properties for grid application programming environments

Grid application programming environments provide both application programming interfaces (APIs) and runtime environments implementing these interfaces, allowing application codes to run in a grid environment. In this section, we outline the properties of such programming environments.

## 2.1 Non-functional properties

We begin our discussion with the non-functional properties as these are determining the constraints on grid API functionality. As such, issues like performance, security, and fault-tolerance have to be taken into account when designing grid application programming environments.

#### Performance

As high-performance computing is one of the driving forces behind grids, performance is the most prominent, non-functional property of the operations that implement the functional properties as outlined below. Job scheduling and placement is mostly driven by expected execution times, while file access performance is strongly determined by bandwidth and latency of the network, and the choice of the data transfer protocol and its configuration (like parallel TCP streams [34] or GridFTP [2]). The trade-off between abstract functionality and controllable performance is a classic since the early days of parallel programming [7]. In grids, it even gains importance due to the large physical distances between the sites of a grid.

#### Fault tolerance

Most operations of a grid API involve communication with physically remote peers, services, and resources. Because of this remoteness, the instabilities of network (Internet) communication, the fact that sites may fail or become unreacheable, and the administrative site autonomy, various error conditions arise. (Transient) errors are common rather

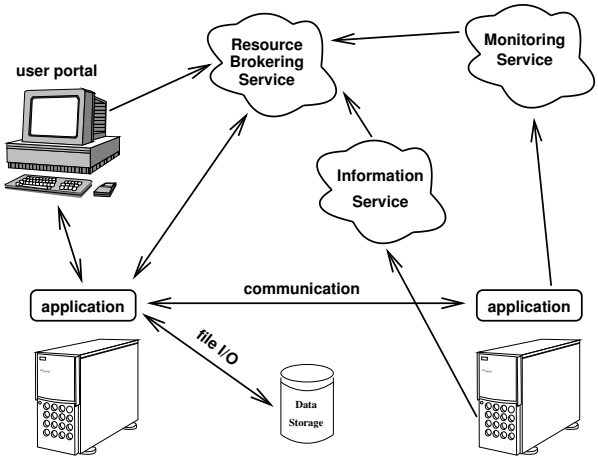



Figure 1: Grid application execution scenario

than the exception. Consequently, error handling becomes an integral part, both of grid runtime environments and of grid APIs.

**Security and trust**

Grids integrate users and services from various sites. Communication is typically performed across insecure connections of the Internet. Both properties require mechanisms for ensuring security of and trust among partners. A grid API thus needs to support mutual authentication of users and resources. Access control to resources (authorization) becomes another source of transient errors that runtime systems and their APIs have to handle. Besides authentication and authorization, privacy becomes important in Internet-based systems which can be ensured using encryption. Whereas encryption need not be reflected in grid APIs themselves, users may notice its presence by degraded communication performance.

**Platform independence**

It is an important property for programming environments to keep the application code independent from details of the grid platform, like machine names or file system layouts for application executables and data files. This needs to be reflected in the APIs provided by a grid programming environment. The corresponding implementations need to map abstract, application-level resources to their physical counterparts.

## 2.2 Functional properties

Figure 1 illustrates the involvement of programming environments in application execution scenarios. We envision the following categories of necessary functionality.

**Access to compute resources, job spawning and scheduling**

Users enter application jobs to the grid via some form of job submission tool, like *globusrun* [39], or a portal like GridSphere [32]. In simple cases, a job will run on a single resource or site. In more advanced scenarios, like dynamic grid applications [3] or in the general case of task-flow applications [5, 37], a running job will spawn off further jobs to available grid resources. But even the portal can be seen as a specialized grid application that needs to submit jobs.

A job submission API has to take descriptions of the job and of suitable compute resources. Only in the simplest cases will the runtime environment have (hard coded) information about job types and available machines. In any real-world grid environment, the mapping and scheduling decision is taken by an external resource broker service [33]. Such an external resource broker is able to take dynamic information about resource availability and performance into account.

#### Access to file and data resources

Any real-world application has to process some form of input data, be it files, data bases, or streams generated by devices like radio telescopes [38] or the LHC particle collider [23]. A special case of input files is the provisioning of program executable files to the sites on which a job has been scheduled. Similarly, generated output data has to be stored on behalf of the users.

As grid schedulers place jobs on computationally suitable machines, data access immediately becomes remote. Consequently, a grid file API needs to abstract from physical file locations while providing a file-like API to the data ("open, read/write, close"). It is the task of the runtime environment to bridge the gap between seemingly local operations and the remotely stored data files.

#### Communication between parallel and distributed processes

Besides access to data files, the processes of a parallel application need to communicate with each other to perform their tasks. Several programming models for grid applications have been considered in the past, among which are MPI [25, 26, 27], shared objects [28], or remote procedure calls [11, 36]. Besides suitable programming abstractions, grid APIs for inter-process communication have to take the properties of grids into account, like dynamic (transient) availability of resources, heterogeneous machines, shared networks with high latency and bandwidth fluctuations. The trade-off between abstract functionality and controllable performance is the crux of designing communication mechanisms for grid applications. Besides, achieving mere connectivity is a challenging task for grid runtime environments, esp. in the presence of firewalls, local addressing schemes, and non-IP local networks [17].

#### Application monitoring and steering

In case the considered grid applications are intended to be long running, users need to be in control of their progress in order to avoid costly repetition of unsuccessful jobs. For this purpose, users need to inspect and possibly modify the status of their application while it is running on some nodes in a grid. For this purpose, monitoring and steering interfaces have to be provided, such that users can interact with their applications. For this purpose, additional communication between the application and external tools like portals or application managers are required.

As listed so far, we consider these properties as the direct needs of grid application programs. Further functionality, like resource lookup, multi-domain information management, or accounting are of equal importance. However, we consider such functionality to be of indirect need only, namely within auxiliary grid services rather than the application programs themselves.

# 3 Existing grid programming environments

After having identified both functional and non-functional properties for grid application programming environments, we now present three existing systems, developed by the authors and their colleagues. For each of them, we outline their purpose and intended functionality, and we discuss which of the non-functional properties can be met. For the three systems, ProActive, Ibis, and GAT, we also outline their architecture and implementation.

## 3.1 ProActive

ProActive is a Java library for parallel, distributed and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API masking

the specific underlying tools and protocols used, and allowing to simplify the programming of applications that are distributed on a LAN, on a cluster of PCs, or on Internet Grids. The library is based on an active object pattern, on top of which a component-oriented view is provided.

**Architecture** All active objects are running in a JVM and more precisely are attached to a *Node* hosted by it. Nodes and active objects on the same JVM are indeed managed by a ProActive runtime (see Figure 2) which provides them support/services, such as lookup and discovery mechanism for nodes and active objects, creation of runtime on remote hosts, enactment of the communications according to the chosen transport protocol, security policies negociation, etc. Numerous meta-level objects are attached to an active object in order to implement features like remote communication, migration, groups, security, fault-tolerance and components. A ProActive runtime inter operates with an open and moreover extensible palette of protocols and tools; for communication and registry/discovery, security: RMI, Ibis, Jini (for environment discovery), web service exportation, HTTP, RMI over ssh tunneling; for process (JVM) deployment: ssh, sshGSI, rsh, Globus (through the JavaCog Kit API), LSF, PBS, Sun Grid Engine. Standard Java dynamic class loading is considered as the means to solve provision of code.

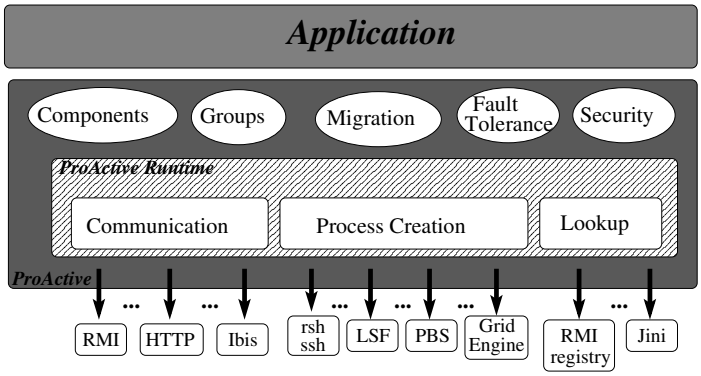



Figure 2: Application running on top of the ProActive architecture

### 3.1.1 Functional properties of ProActive

**Access to computing resources, job spawning and scheduling** Instead of having an API which mandates the application to configure the infrastructure it must be deployed on, deployment descriptors are used. They collect all the deployment information required when launching or acquiring already running ProActive runtimes (e.g. remote login information, CLASSPATH value) and ProActive proposes the notion of a Virtual Node which serves to virtualize the active object's location in the source code (see figure 3). Besides, the mapping from a virtual node to effective JVMs is managed via these deployment descriptors [9]. In summary, ProActive provides an open deployment system (through a minimum size API, associated with XML deployment files) that enables to access and launch JVMs on remote computing resources using an extensible palette of access protocols. ProActive provides a simple API to trigger (weak) migration of active objects as a means to dynamically remap activities on the target infrastructure (e.g. `ProActive.migrateTo(Node anode)`).

**Communication for parallel processes** Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [14]. Each active object has its own thread of control, and decides in which order to serve the incoming method call requests. Based on a simple Meta-Object Protocol, a communication between those active objects follows the standard Java method invocation syntax. Thus no specific API is required in order to let them communicate. ProActive provides an extension of this to groups of active objects, as a *typed group communication* mechanism. On a Java class, named e.g. `A`, here is an example of a typical group creation and method call

```
// A group of type "A" and its 3 members are created at once on the nodes
```

```
ProActiveDescriptor pad = ProActive.getProActiveDescriptor(String xmlFileLocation);
//---- Returns a ProActiveDescriptor object from the xml file
VirtualNode dispatcher = pad.getVirtualNode("Dispatcher");
//---- Returns the VirtualNode Dispatcher described in the xml file as an object
dispatcher.activate()
// --- Activates the VirtualNode
Node node = dispatcher.getNode();
//-----Returns the first node available among nodes mapped to the VirtualNode
C3DDispatcher c3dDispatcher = ProActive.newActive("org.objectweb.proactive.core.
examples.c3d.C3DDispatcher", param, node);
//-----Creates an active object running class C3DDispatcher, on the remote JVM.
```

Figure 3: Example of a ProActive source code for descriptor-based mapping

```
// directly specified, parameters are specified in params,
Object[][] params = {{...}, {...}, {...}};
A ag = (A) ProActiveGroup.newGroup("A",params, {node1,node2,node3});
V vg=ag.foo(pg); // A typed group communication
```

The *result* of a typed group communication, which may also be a group, is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results. Other features are provided through methods of the group API: parameter dispatching instead of broadcasting, using *scatter* groups, explicit group method call synchronization through futures (e.g. `waitOne`, `waitAll`), dynamic group manipulation by first getting the group representation, then `add`, `remove` of members. Groups provide an object oriented SPMD programming model [6].

**Application monitoring and steering** ProActive applications can be monitored transparently by an external application, written in ProActive: IC2D (Interactive Control and Debugging for Distribution) [9]. Once launched, it is possible to select some hosts (and implicitly all corresponding virtual nodes, nodes, and hosted active objects): this triggers the registration of a listener of all sort of events that occur (send of method calls, reception of replies, waiting state); they are sent to IC2D, then graphically presented. IC2D provides a drag-and-drop migration of active objects from one JVM to an other, which can be considered as a steering operation. A job abstraction enables to consider at once the whole set of activities and JVMs that correspond to the specific execution of an application. Graphical job monitoring is then provided, e.g. to properly kill it.

### 3.1.2 Non-functional properties of ProActive

**Performance** As a design choice, communication is asynchronous with futures. Compared to traditional futures, (1) they are created implicitly and systematically, (2) and can be passed as parameters to other active objects. As such, performance may come from a good overlapping of computation and communication. If pure performance is a concern, then ProActive should preferably use Ibis instead of standard RMI, as the transport protocol [24].

**Fault tolerance** Non functional exceptions are defined and may be triggered, for each sort of feature that may fail due to distribution. Handlers are provided to transparently manage those non-functional exceptions, giving the programmer the ability to specialize them. Besides, a ProActive application can transparently be made fault-tolerant. On user demand, a transparent checkpointing protocol, designed in relation with an associated recovery protocol, can be applied. Once any active object is considered failed, the whole application is restarted from a coherent global state. The only user's involvement is to indicate in the deployment descriptor, the location of a stable storage for checkpoints.

**Security and trust** The ProActive security framework allows to configure security according to the deployment of the application. Security mechanisms apply to basic features like communication authentication, integrity, confidentiality to more high-level ones like object creation or migration (e.g. a node may accept or deny the migration of an active object according to its provenance). Security policies are expressed outside the application, in a security descriptor attached to the deployment descriptor that the application will use. Policies are negotiated dynamically by the participants involved in a communication, a migration, or a creation, be they active objects or nodes, and according

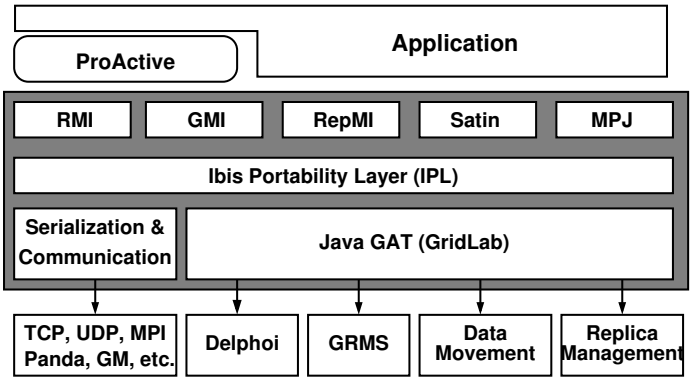



Figure 4: API's and architecture of the Ibis system

to their respective IP domain and ProActive runtime they are hosted by. The security framework is based on PKI. It is possible to tunnel over SSH all communications towards a ProActive runtime, as such multiplexing and demultiplexing all RMI connections or HTTP class loading requests to that host through its ssh port. For users, it requires only to specify in ProActive deployment descriptors, for instance, which JVMs should export their RMI objects through a SSH tunnel.

**Platform independence** ProActive is built in such a way as it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

## 3.2 Ibis

The Ibis Grid programming environment [42] has been developed to provide parallel applications with highly efficient communication API's. Ibis is based on the Java programming language and environment, using the "write once, run anywhere" property of Java to achieve portability across a wide range of Grid platforms. Ibis aims at Grid-unaware applications. As such, it provides rather high-level communication API's that hide Grid properties and fit into Java's object model.

The Ibis runtime system architecture is shown in Figure 4. Ibis can be configured dynamically at run time, allowing to combine standard techniques that work "anywhere" (e.g., using TCP) with highly-optimized solutions that are tailored for special cases, like a local Myrinet interconnect. The *Ibis Portability Layer* (IPL), that provides this flexibility, consists of a small set of well-defined interfaces. The IPL can have different implementations, which can be selected and loaded into the application *at run time*.

The IPL allows configuration via properties (key-value pairs), like for the serialization method that is used, reliability, message ordering, performance monitoring support, etc. Whereas the layers on top of IPL request certain properties, the Ibis instantiation is using local configuration files containing information about the locally available functionality (like being reliable or unreliable, or having ordered broadcast communication). At startup, Ibis tries to load each of the implementations listed in the file, and checks if they adhere to the required properties, until all requirements have been met. If this is impossible, properties are re-negotiated with the layers on top of IPL.

### 3.2.1 Non-functional properties of Ibis

**Performance**
For Ibis, performance is the paramount design criterion. The IPL provides communication primitives using send ports and receive ports. A careful design of these ports and primitives allows flexible communication channels, streaming of data, efficient hardware multicast and zero-copy transfers.

The layer above the IPL creates send and receive ports, which are connected to form a *unidirectional message channel*, see Figure 5. New (empty) message objects can be requested from send ports, and data items of any type can

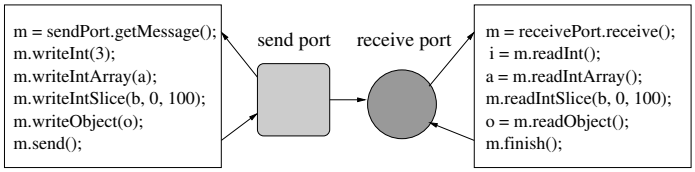



Figure 5: Ibis send ports and receive ports

be inserted incrementally, allowing for streaming of long messages. Both primitive types and arbitrary objects can be written. When all data is inserted, the *send* primitive can be invoked on the message.

The IPL offers two ways to receive messages, the port's blocking *receive* primitive, as well as upcalls, providing the mechanism for implicit message receipt.

**Fault tolerance**
Currently, the Ibis runtime system provides transparent fault tolerance for the Satin divide-and-conquer API (see below) [45]. Fault tolerance for other programming models is subject to ongoing work.

**Security and trust**
Ibis is focusing on communication between processes. Currently, it supports encrypted communication by incorporating the secure socket layer (SSL) in its communication protocol stack [17]. Issues like authentication and authorization are beyond the scope of Ibis.

**Platform independence**
Platform independence is mostly addressed by using Java. Besides, the IPL's dynamic loading facility hides many properties of the underlying resources and networks from the application.

### 3.2.2 Functional properties of Ibis

**Access to compute resources, job spawning and scheduling**
The API's provided by Ibis focus on communication between parallel processes. As such, they do not provide access to compute resources, job spawning, or scheduling systems. Instead, Ibis assumes that underlying Grid middleware takes care of scheduling and starting an Ibis application. The Ibis model does not prevent settings with processes joining and leaving running applications, however, such dynamically changing process groups are not exposed to the application.

**Access to file and data resources**
As Ibis is hiding the Grid environment from the application, its API's do not provide access to Grid file and data resources either. As with compute resources, Ibis assumes underlying middleware (like the Java GAT [4]) to provide such functionality, if needed.

**Communication between parallel and distributed processes**
As shown in Figure 4, Ibis provides a set of communication API's on top of the IPL. Besides the API's shown in the figure and discussed below, Ibis also supports CCJ [31], a simple set of collective communication operations, inspired by MPI [30]. An implementation of the MPJ [15] message passing API has been added recently. Support for GridSuperscalar [5] is subject to ongoing work.

**RMI** For basic, object-based communication, Ibis provides Java's standard *remote method invocation* (RMI) [42]. The Ibis RMI implementation is optimized for high performance between remote processes [42].

**RepMI** Using RMI for globally shared objects can lead to serious performance degradation as almost all method invocations have to be performed remotely across a Grid network. For applications with high read/write ratios to their shared objects, replicated objects can perform better, as all read operations become local, and only write operations need to be broadcast to all replicas, as implemented in *replicated method invocation* (RepMI) [28].

With RepMI, groups of replicated objects are identified by the programmer using special marker interfaces (like with Java RMI). The Ibis runtime system works together with a byte code rewriter (similar to RMI's *rmic*) that understands the special marker interfaces and generates communication code for the respective classes.

**GMI** Compared to RepMI, *group method invocation* (GMI) [29] can provide more relaxed consistency among *object groups*, favoring higher performance and much better flexibility. GMI is a conceptual extension of RMI, also allowing groups of *stubs* and *skeletons* to communicate, allowing RMI-like *group* communication. In GMI, methods can be invoked either on a single stub, or collectively on a group of stubs. Likewise, invocations can be handled by an individual skeleton or a group of skeletons. Complex communication patterns among stubs and skeletons can be deployed, depending on the communication semantics. Schemes for method invocation and result handling can be combined orthogonally, providing a wide spectrum of operations that span, among others, both synchronous and asynchronous RMI, future objects, and collective communication as known from MPI.

**Satin** Divide-and-conquer parallelism is provided using the Satin interface [41], shown in Figure 6. The application first extends the *Spawnable* marker interface, indicating to the byte code rewriter to generate code for parallel execution. An application class then extends the *SatinObject* class and implements its marker interface, together tagging the recursive invocations as asynchronous. The *sync* method blocks until all spawned invocations have returned their result.

```
interface FibInterface extends ibis.satin.Spawnable {
          public long fib(long n);
}

class Fib extends ibis.satin.SatinObject implements FibInterface {
      public long fib(long n) {
             if(n < 2) return n;
             long x = fib(n-1); /* spawn, tagged in FibInterface */
             long y = fib(n-2); /* spawn, tagged in FibInterface */
             sync();            /* from ibis.satin.SatinObject   */
             return x + y;
      }
}
```

Figure 6: Satin code example for the Fibonacci numbers

Each spawned method invocation creates an invocation record that is stored locally in a work queue. Idle processors obtain work by an algorithm called *cluster-aware random work stealing* (CRS). It has been shown in [43] that CRS can execute parallel applications very efficiently in Grid environments while the application code is completely shielded from Grid peculiarities by the Satin interface.

**Application monitoring and steering** Application monitoring and steering are not provided explicitly by Ibis; this is subject to ongoing work.

## 3.3 GAT

The Grid Application Toolkit (GAT) aims to enable scientific applications in grid environments. It helps to integrate grid capabilities in application programs, by providing a simple and *stable* API with well known API paradigms (e.g. POSIX like file access), interfacing to grid resources and services, abstracting details of underlying grid middleware. This allows to interface to different versions or implementations of grid middleware without any code change in the application.

To illustrate how the GAT can be used, Figure 7 shows a complete C++ program performing a remote file access – it reads the file given as command line argument and prints its content on *stdout*.

```
#include <iostream>
#include <GAT.hpp>
int main (int argc, char** argv) {
  try {
    GAT::Context    context;
    GAT::FileStream file (context, GAT::Location (argv[1]));
    char buffer[1024] = "";

    while ( 0 < file.Read (buffer, sizeof (buffer) - 1) )
      std::cout << buffer;
  }
  catch (GAT::Exception const &e) {
    std::cerr << e.GetMessage() << std::endl;
  }
  return (0);
}
```

Figure 7: GAT example in C++ for a `cat`.

To the application, the file access method actually used is completely transparent – it could be ssl, ftp, grid-ftp or libc. The GAT takes care of translating these calls to the appropriate middleware operations, by preserving the defined semantics.

### 3.3.1 Functional properties of GAT

The GAT API specification covers the six following areas:

(1) physical files
(2) logical files
(3) communication
(4) compute resources
(5) monitoring and steering
(6) persistent meta data and information

These areas have been derived from application use cases in the GridLab [3] project, and are not intended to provide complete coverage of grid capabilities. Hence, GAT is not intended to cover *every* application use case in grids (although the GAT authors tried to extend the scope to other probable use cases).

*Simplicity* is *the* major constraint for the API specification, along with the requirement to reuse well known API paradigms wherever possible. These constraints lead to the API as described in more detail in [4].

The GAT API addresses all functional properties as listed in Section 2.2. However, there is one notable exception: GAT does not provide any high-performance inter process communication. GAT's communication mechanisms are aimed at application steering and control and support only simple pipes. High-performance communication between parallel processes is considered to be beyond the scope of GAT.

**Architecture** The GAT architecture [4] follows 2 major design goals: (a) the API layer is to be *independent* from the grid environment; and (b) bindings to the grid environments must be exchangeable/extendable at runtime, on user and/or administrator level. This implies that the API layer cannot implement the API capabilities directly, but has to dynamically dispatch the calls to a lower, exchangeable layer. That principle is reflected in the API as shown in Figure 8.

A thin API layer with GAT syntax interfaces to the application. The calls are forwarded to the GAT engine, which dynamically dispatches the API calls to the adaptor layer. Adaptors are modules which implement the semantics of the call and bind the GAT to specific a grid middleware.

Recently, the Global Grid Forum (GGF) has formed a research group (SAGA-RG – Simple API for Grid Applications) to standardize a grid API. The GAT group is actively taking part in this group. In fact, the SAGA design will be very similar to the GAT design. However, SAGA prescribes only the API, not the architecture of any implementation.

### 3.3.2 Non-functional properties of GAT

**Performance** In grid environments, network latency and remote service delays form a major performance problem. Compared to that, any overhead imposed by the local implementation is small. Hence, the GAT engine's dynamic system of adaptor selection and call dispatching introduces little overhead, when compared to the total call execution

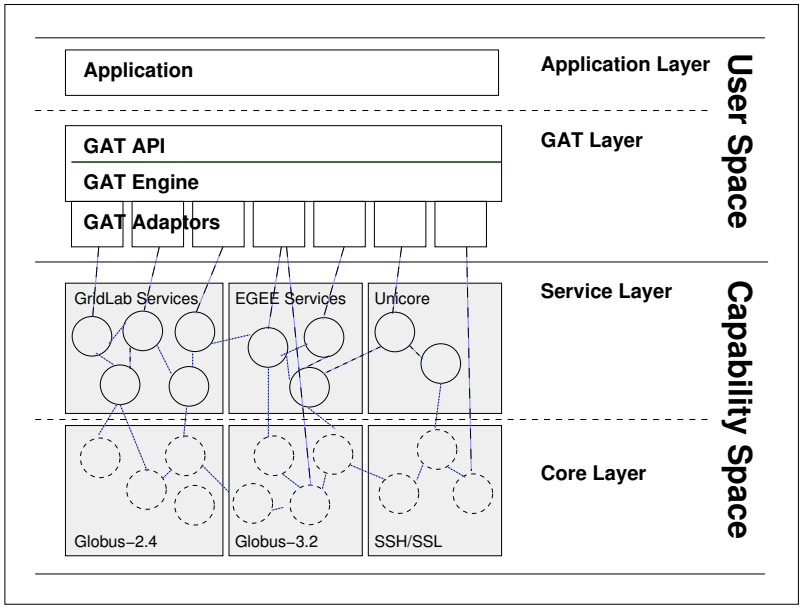



Figure 8: GAT Architecture: the engine dispatches API calls to adaptors, which bind to a specific grid capability provider.

time. However, GAT can be configured to make optimal use of available resources. For example, a system administrator may want to install/configure a file movement adaptor using a locally-available high-speed LAN. Also, as in all grid applications, adaptor-level caching mechanisms are encouraged to minimize network latencies.

**Fault Tolerance** Upon failure of a grid operation, the GAT engine is falls back to other adaptors. For example, if a GSI adaptor fails to create a coomunication channel, a SSH adaptor would be tried, and may succeed. That process is transparent to the application.

By abstracting the grid operations from the application, fault tolerance is much easier to provide, and can be hidden from the end user. But error reporting and auditing are crucial for the user. GAT supports both, and returns a hierarchical stack trace of call information for all operations performed.

On adaptor level, fault tolerance is implemented in whichever manner is suitable for the grid environment the adaptor binds to.

**Security and Trust** GAT can only be as secure as its adaptors – the engine itself does not perform remote operations and cannot directly enforce any security. However, the engine provides the means to implement a coherent security model across all adaptors. Also, a minimal security level (`local, ssh, gsi` etc.) can be specified via user and system preferences, and prevents less secure adaptors from being used. Ultimately, the trust relationship lies with the adaptors, and with the grid middleware, rather than with GAT. GAT merely delegates, capabilities as well as security.

**Platform Independence** Platform, environment and language independence has been a major objective of the GAT implementation. The GAT reference implementation is written in ANSI-C/ISO-C99, is natively developed on Windows and Linux, and has been ported to MacOS-X, Solaris, IRIX, Linux-64 and other UNIX's. A native Java GAT implementation is also available, which by design is platform independent.

The GAT architecture makes it usable on any grid environment providing a set of adaptors. GAT comes with a set of local adaptors, binding the API to the *libc* – hence it is possible to develop complex grid applications on disconnected systems, and to later run the same executable on a full scale grid.

GAT adaptors exist to GridLab services [3], to Globus (pre Web Services) [39], to ssh/ssl/sftp, and to the Unicore Resource Broker [18]. Ongoing work is addressing adaptor development for specific experiments or projects, like to the Storage Resource Broker (SRB) [8], or for dynamic light-path allocation for file transfer. The Java GAT implementation uses the CoG Toolkit [44] to bind to various Globus incarnations.

### 3.4 Summary

We summarize the comparison of our three systems in Table 1. There, bullet points indicate properties that are addressed while hollow circles refer to unaddressed properties. From the table it becomes obvious that the three systems have been designed for somewhat different purposes. These choices directly influence which (functional and non-functional) properties are addressed, actually.

| Property | ProActive | Ibis | GAT |
|---|---|---|---|
| *Non-Functional Properties* | | | |
| performance | ● | ● | ○ |
| fault tolerance | ● | ● | ● |
| security / trust | ● / ○ | ● / ○ | ● / ○ |
| platform independence | ● | ● | ● |
| *Functional Properties* | | | |
| resources / job spawning / scheduling | ● / ● / ○ | ○ / ○ / ○ | ● / ● / ● |
| files / data resources | ○ / ○ | ○ / ○ | ● / ● |
| parallel / distributed communication | ● / ● | ● / ● | ○ / ● |
| application monitoring / steering | ● / ○ | ○ / ○ | ● / ● |

Table 1: Comparison of the three frameworks

## 4 Related work

Besides the ones presented, there are various other grid programming environments. First of all, one has to name the "native" grid APIs and environments, such as Condor [20] and Globus [39]. These systems reach deep into the fabric layer of the grid, but also provide programming interfaces for higher levels. However, these interfaces are often conceived as being unfit for application development, in terms of complexity and dependency on the actual middleware and its configuration.

The Java CoG [44] has been an early valiant effort to offer low level grid capabilities to applications. First, CoG has been a wrapper around early Globus versions. The CoG became a reimplementation of large parts of Globus, to get independent of the Globus development cycle. Nowadays, CoG has a very similar architecture to GAT, and to the general architecture presented in Section 5. CoG is, as GAT, one of the major supporters for the GGF SAGA API effort.

Early versions of the Java CoG have been binding directly and exclusively to a specific Globus version. In fact, that approach was taken in other projects as well, with the main objective to abstract the complexity of grid programming for the end user. However, that approach can not be kept in sync with neither the dynamic grid environment, nor with the progress of the grid middleware development.

Various MPI implementations aim at supporting large parallel applications in grids. MPICH-G [19] is using Globus communication facilities. PACX-MPI [26] is an MPI implementation which can efficiently handle WAN connectivity. Both packages (and others with the same scope) are widely used in the community.

Remote steering has been an interesting target for distributed applications for a long time. It seems to be one of the areas which could benefit significantly from the use of grid paradigms – the absence of a suitable distributed security framework has hindered the widespread use of remote steering until now. The Reality Grid Project [13] is providing one example of a grid-based steering infrastructure, based on the concepts of OGSA and Web Services. The Reality Grid Project is also one of the supporters of the GGF SAGA API.

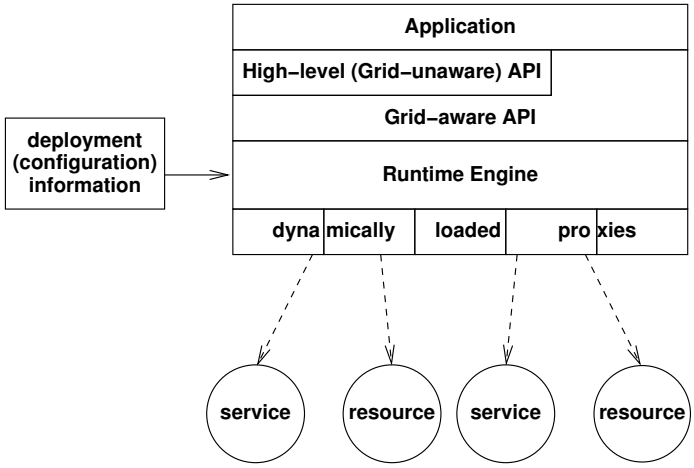



Figure 9: Generic runtime architecture model

ICENI [21] is a grid middleware framework with an added value to the lower-level grid services. It is a system of structured information that allows to match applications with heterogenous resources and services, in order to maximise utilisation of the grid fabric. Applications are encapsulated in a component-based manner, which clearly separates the provided abstraction and its possibly multiple implementations. Implementations are selected at runtime, so as to take advantage of dynamic information, and are selected in the context of the application, rather than a single component. This yields to an execution plan specifying the implementation selection and the resources upon which they are to be deployed. Overall, the burden of code modification for specific grid services is shifted from the application designer to the middleware itself.

# 5 Generic architecture model

Our three systems, ProActive, Ibis, and GAT, provide API functionality that partially overlaps, and partially complements each other. A much stronger similarity, however, can be observed from their software architectures, which is due to the non-functional properties of platform independence, performance, and fault tolerance. These properties strongly call for systems that are able to dynamically adjust themselves to the actual grid environment underneath.

Figure 9 shows the generic architecture for grid application programming environments that can address the properties identified in Section 2.

- Application code is programmed exclusively using the API's provided by the envisioned grid application programming environment. We distinguish between grid-aware (lower level) and grid-unaware (higher level) API's. Both kinds of API's will be needed, depending on the purpose of the application. For example, one kind of application might simply wish to use some invisible computational "power" provided by the grid (as intended by the power grid metaphor) while others might want to interact explicitly with specific resources like given databases or scientific instruments.

- The API's are implemented by a *runtime engine*. The engine's most important task is to delegate API invocations to the right service or resource. In the presence of transient errors and of varying performance, this delegation becomes a non-trivial task. In fact, the runtime engine should implement sophisticated strategies to bind requests to the best-suited service. Whereas possibly many services might fulfill a given request, the choice among them is guided purely by non-functional properties like service availability, performance, and security level.

- Delegation to a selected service can be achieved by dynamically loaded proxies. Dynamic binding is important for separating application programs from the actual grid execution environments. This way, applications can also be executed in new (versions of) environments that came to existence only after an application has been

written. Besides for platform independence, this dynamic binding is necessary for handling of transient error conditions at runtime.

- For resource and service selection purposes, the runtime engine needs configuration information to provide the right bindings. Current implementations (like ProActive, Ibis, and GAT) mostly rely on configuration files, provided by system administrators, describing the properties of given machines in a grid. However, some configuration information is of more dynamic nature and requires dynamic monitoring and information services to provide up-to-date information. Examples are access control for users and resources, and performance data about network connections that might impact resource selection.

It is obvious that current grid application programming environments comply only partially to this architecture. However, with the advent of more sophisticated grid middleware, like grid component architectures, or widely deployed monitoring and information services, also programming environments will be able to benefit and provide more flexible, better performing, and failure-resilient services to applications.

# 6 Conclusions and future directions

Grids can be considered as distributed systems for which heterogeneity, wide-area distribution, security and trust requirements, failure probability, as well as latency and bandwidth of communication networks are exacerbated. Such platforms currently challenge application programmers and users. Tackling these challenges calls for significantly advanced application programming environments.

We have identified a set of functional and non-functional properties of such future application programming environments. Based on this set, we have analyzed existing environments, emphasizing ProActive, Ibis, and GAT, which have been developed by the authors and their colleagues, which we also consider to be among the currently most advanced systems.

Our analysis has shown that none of our systems curently addresses all properties. This is mostly due to the different application scenarios for which our systems have been developed. Based on our analysis, we have identified a generic architecture for future grid programming environments that allows building systems that will be capable of addressing the complete set of properties, and will thus be able to overcome today's problems and challenges.

A promising road to implementing our envisioned grid application programming environment is to explore a component-oriented approach, as also proposed in [40]. To date, GridKit [16] seems to be unique in effectively applying such a component-based approach to both the runtime environment and the application layer of a grid platform. Most existing component-based systems address applications only, like [1, 21] or the implementation of Fractal components using ProActive [10].

The inherent openness, introspection and reconfiguration capabilities offered by a component-oriented approach appear promising for implementing both grid programming environments and applications that are portable, adaptive, self-managing, and self-healing. Implementing such properties will require advanced decision taking and planning inside the runtime engine. Applying a component-oriented approach will thus complement the generic architecture identified in this chapter. Components will be the building blocks that ae assembled and reassembled at run time, yielding flexible grid application environments.

# acknowledgments

This manuscript would not have been possible without the many contributions of our past and present colleagues. We would like to thank all the major contributors in the design and development of ProActive: by contribution order, J. Vayssière, L. Mestre, R. Quilici, L. Baduel, A. Contes, M. Morel, C. Delbé, A. di Costanzo, V. Legrand, G. Chazarain. We owe a lot to the Ibis team: Jason Maassen, Rob van Nieuwpoort, Ceriel Jacobs, Rutger Hofman, Kees van Reeuwijk, Gosia Wrzesinska, Niels Drost, Olivier Aumage, and Alexandre Denis. We also express our thanks to the GAT designers and writers. GAT was designed by Tom Goodale, Gabrielle Allen, Ed Seidel, John Shalf and others. The C Version has mainly been implemented by Hartmut Kaiser, who also wrote the C++ and Python wrappers. The Java version was written by Rob van Nieuwpoort. We would like to thank our many colleagues from the EU GridLab project.

The current collaboration is carried out in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (contract IST-2002-004265). ProActive is supported by INRIA, CNRS, French Ministry of

Education, DGA, through PhD funding, and ObjectWeb, ITEA OSMOSE, France Telecom R&D. Ibis is supported by the Netherlands Organization for Scientific Research (NWO) and the Dutch Ministry of Education, Culture and Science (OC&W), and is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). GAT has been supported via the European Commission's funding for the GridLab project (contract IST-2001-32113).

# References

[1] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured iplementation of component based grid programming environments. in this volume.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. GGF GridFTP Working Group Document, 2002.

[3] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, Andre Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid - A GridLab Overview. *International Journal on High Performance Computing Applications*, 17(4):449–466, 2003.

[4] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.

[5] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[6] L. Baduel, F. Baude, and D. Caromel. Object-Oriented SPMD. In *CCGrid 2005*, 2005.

[7] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[8] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[9] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In *HPDC-11*, pages 93–102. IEEE Computer Society, July 2002.

[10] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *DOA*, volume 2888, pages 1226–1242. LNCS, 2003.

[11] M. Beck, J. Dongarra, J. Huang, T. Moore, and J. Plank. Active Logistical State Management in the GridSolve/L. In *Proc. 4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2004.

[12] R. Berlich, M. Kunze, and K. Schwarz. Grid Computing in Europe: From Research to Deployment. *CRPIT series, Proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid 2005)*, 44, Jan. 2005.

[13] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational Steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting 2003*, 2003.

[14] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[15] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurency: Practice and Experience*, 12(11):1019–1038, 2000.

[16] G. Coulson, P. Grace, P. Blair, and al. Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing. *To appear in Concurrency and Computation: Practice and Experience*, 2005.

[17] Alexandre Denis, Olivier Aumage, Rutger Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *Proc.HPDC-13*, pages 97–106, 2004.

[18] Dietmar Erwin, editor. *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.

[19] Ian Foster and Nicholas T. Karonis. A grid-enabled mpi: message passing in heterogeneous distributed computing systems. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society, 1998.

[20] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[21] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T . Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12), 2002.

[22] The Global Grid Forum (GGF). http://www.gridforum.org/.

[23] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *Proc. IEEE/ACM International Workshop on Grid Computing (Grid'2000)*, 2000.

[24] Fabrice Huet, Denis Caromel, and Henri E. Bal. A High Performance Java Middleware with a Real Application. In *SuperComputing 2004*, 2004.

[25] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[26] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Towards efficient execution of MPI applications on the Grid: porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003.

[27] Thilo Kielmann, Rutger F.H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A.F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, 1999.

[28] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8–9):681–712, 2001.

[29] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Proc. LCR '02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington, DC, 2002. To be published in LNCS.

[30] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[31] Arnold Nelisse, Jason Maassen, Thilo Kielmann, and Henri E. Bal. CCJ: Object-based Message Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3–5):341–369, 2003.

[32] Jason Novotny, Michael Russell, and Oliver Wehrens. GridSphere: A Portal Framework for Building Collaborations. In *1st International Workshop on Middleware for Grid Computing*, Rio de Janeiro, 2003.

[33] Jennifer M. Schopf, Jarek Nabrzyski, and Jan Weglarz, editors. *Grid resource management: state of the art and future trends*. Kluwer, 2004.

[34] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Proc. Supercomputing (SC2000)*, 2000.

[35] O. Smirnova, P. Eerola, T. Ekelof, M. Elbert, J.R. Hansen, A. Konstantinov, B. Konya, J.L. Nielsen, F. Ould-Saada, and A. Waananen. The NorduGrid Architecture and Middleware for Scientific Applications. In *ICCS 2003*, number 2657 in LNCS. Springer-Verlag, 2003.

[36] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[37] Ian Taylor, Matthew Shields, Ian Wang, and Omer Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[38] The GEO600 project. http://www.geo600.uni-hannover.de/.

[39] The Globus Alliance. http://www.globus.org/.

[40] J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards Building a Generic Grid Services Platform: a component-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*. Springer Verlag, 2005.

[41] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *Proc. PPoPP '01: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, 2001.

[42] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, 2005.

[43] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Thilo Kielmann, and Henri E. Bal. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. *Journal of Supercomputing*, accepted for publication, 2004.

[44] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[45] Gosia Wrzesinska, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal. Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, USA, 2005.

## 2.3 Programmation par composants

### 2.3.1 From distributed objects to hierarchical grid components.

F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, LNCS, pages 1226–1242. Springer Verlag, 2003.

# From Distributed Objects to Hierarchical Grid Components

Françoise Baude, Denis Caromel, and Matthieu Morel

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia-Antipolis
2004, Route des Lucioles, BP 93
F-06902 Sophia-Antipolis Cedex - France
{Francoise.Baude,Denis.Caromel,Matthieu.Morel}@sophia.inria.fr

**Abstract.** We propose a parallel and distributed component framework for building Grid applications, adapted to the hierarchical, highly distributed, highly heterogeneous nature of Grids. This framework is based on ProActive, a middleware (programming model and environment) for object oriented parallel, mobile, and distributed computing. We have extended ProActive by implementing a hierarchical and dynamic component model, named Fractal, so as to master the complexity of composition, deployment, re-usability, and efficiency of grid applications. This defines a concept of Grid components, that can be parallel, made of several activities, and distributed. These components communicate using typed one-to-one or collective invocations.

**Keywords:** Active objects, components, hierarchical components, grid computing, deployment, dynamic configuration, group communications, ADL.

## 1 Introduction

In this article, we present a contribution to the problem of software reuse and integration for distributed and parallel object-oriented applications. We especially target grid-computing. Our approach takes the form of a programming and deployment framework featuring parallel, mobile and distributed components, so our application domains also target mobile and ubiquitous distributed computing on the Internet (where high performance, high availability, ease of use, etc., are of importance).

For Grid applications development, there is indeed a need also to smoothly, seamlessly and dynamically integrate and deploy autonomous software, and for this provide a *glue* in the form of a software bus. In this sense, we essentially address the second group of Grid programmers such as defined in [1]: first group of users are end users who program pre-packaged Grid applications by using a simple graphical or Web interface; the second group of grid programmers are those that know how to build Grid applications by composing them from existing application "components"; the third group consists of the researchers that build the individual components.

R. Meersman et al. (Eds.): CoopIS/DOA/ODBASE 2003, LNCS 2888, pp. 1226–1242, 2003.

We share the goal of providing a component-based high-performance computing solution with several projects such as: CCA [1] with the CCAT/XCAT toolkit [2] and Ccaffeine framework, Parallel CORBA objects [3] and GridCCM [4]. But, to our knowledge, our contribution is the first framework featuring *hierarchical* distributed components. This clearly helps in mastering the complexity of composition, deployment, re-usability required when programming and running large-scale distributed applications.

We propose a parallel and distributed component framework for building meta-computing applications, that we think is well adapted to the hierarchical, highly distributed, highly heterogeneous nature of grid-computing. This framework is based on ProActive, a Java-based middleware (programming model and environment) for object oriented parallel, mobile and distributed computing. ProActive has proven to be relevant for grid computing [5] especially due to its deployment and monitoring aspects [6] and its efficient and typed collective communications [7]. We have succeeded in defining a component model for ProActive, with the implementation of the Fractal component model [8,9], mainly taking advantage of its hierarchical approach to component programming.

Fractal is a general software composition model, implemented as a framework that supports component-based programming, including hierarchical components (type) definition, configuration, composition and administration. Fractal is an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. Indeed, a system administrator, a system integrator or an application developer may need to dynamically construct a system or service out of existing components, whether in response to failures, as part of the continuous evolution of a running system, or just to introduce new applications in a running system (a direct generalization of the dynamic binding used in standard distributed client-server applications). Nevertheless, the requirements raised by distributed environments are not specifically addressed by the Fractal model. Not because this is not an issue, but, because, according to the Fractal specification, a primitive or hierarchical fractal component *may be* a parallel and distributed software. So, our work also yields to a new implementation of the Fractal model that explicitly provides parallel and distributed Fractal components.

The main achievement of this work is to design and implement a concept of *Grid Components*. Grid components are recursively formed of either sequential, parallel and/or distributed sub-components, that may wrap legacy code if needed, that may be deployed but further reconfigured and moved – for example to tackle fault-tolerance, load-balancing, adaptability to changing environmental conditions.

Below is a typical scenario illustrating the usefulness of our work. Assume a complex grid software be formed of several services, say of other software (a parallel and distributed solver, a graphical 3D renderer, etc). The design of such a software is very much simplified if it can be considered as a hierarchical composition (recursive assembly and binding): the solver is itself a component composed of several components, each encompassing a piece of the computation; the whole software is seen as a single component formed of the solver and the

renderer. From the outside, the usage of this software is as simple as invoking a functional service of a component (e.g. call *solve-and-render*). Once deployed and running on a grid, assume that due to load balancing purposes, this software needs to be relocated. Some of the on-going computations may just be moved (the solver for instance), alas others depending on specific peripherals that may not be present at the new location (the renderer for instance) may be terminated and replaced by a new instance adapted to the target environment and offering the same service. As the solver is itself a hierarchical component formed of several sub-components, each encompassing an activity, we trigger the migration of the solver as a whole, without having to explicitly move each of its sub-components, while references towards mobile components remain valid. And once the new graphical renderer is launched, we re-bind the software, so as it now uses this new instance.

This paper is organized as follows: after an introduction on parallel and distributed programming with ProActive, and on the Fractal component model, the principles and design of the proposed parallel and distributed component model are presented. The implementation and an example are described in section 4, while section 5 makes a comparison with related work before concluding.

## 2 Context

### 2.1 Distribution, Parallelism, Mobility, and Deployment with ProActive

The ProActive middleware is a 100% Java library (LGPL) [10] aiming to achieve seamless programming for concurrent, parallel, distributed and mobile computing. The main features regarding the programming model are:

- a uniform *active object* programming pattern
- remotely accessible objects, via method invocation
- asynchronous communications with automatic synchronization (automatic futures for results of remote method calls). Note that asynchronicity enables to use one-way calls for transmitting events.
- group communications, which enable to trigger method calls on a distributed group of active objects of the same compatible type, with a dynamic generation of groups of results. It has been shown in [7] that this group communication mechanism, plus a few synchronization operations (`WaitAll, WaitOne`, etc), provides quite similar patterns for collective operations such as those available in e.g. MPI, but in a language centric approach. Here is an example:

```
//Object 'a' of class A is an active remote object
V v = a.foo(param);
// remotely calls foo on object a

v.bar();
// automatically blocks on v.bar()
// until the result in v gets available.
```

```
// ag is a group of active objects,
// of types compatible with A
V v = ag.foo(param);
// calls foo on each group member
// with some optimisation at serialization time
// V is automatically built as a group of results
v.bar();
// executes as soon as individual results
// of foo calls return
```

– migration (mobile computations): An active object with its pending requests (method calls), its futures, its passive (mandatory non-shared) objects may migrate from JVMs to JVMs. The migration may be initiated from outside through any public method but it is the responsibility of the active object to execute the migration (weak migration). Automatic, transparent (and optimized) forwarding of requests and replies provide location transparency, as remote references towards active mobile objects remain valid.

We are faced with the common difficulties in deployment regarding launching a ProActive application in its environment. We succeed in completely avoid scripting for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code such as to gain in flexibility [6] as follows (see figure 8 for an example):

– XML Deployment Descriptors. Active objects are remotely created on JVMs, but *virtual nodes* are manipulated inside the program, instead of URLs of JVMs. The *mapping* of virtual nodes to effective JVMs is managed externally through those descriptors. Descriptors also permit to define how to launch JVMs.
– Interfaces with various protocols: `rsh`, `ssh`, `LSF`, `Globus`, `Jini`, `RMIregistry` enable to effectively launch, register or discover JVMs according to the needs specified in the descriptor.
– Graphical visualization and monitoring of any ongoing ProActive application is possible through a ProActive application called *IC2D* (Interactive Control and Debugging of Distribution). In particular, IC2D enables to migrate executing tasks by a graphical drag-and-drop, and to create additional JVMs.

### 2.2 The Fractal Component Model

The Fractal component model provides an homogeneous vision of software systems structure with a few but well defined concepts such as component, controller, content, interface, binding. It also exhibits distinguishing features that have proven useful for the present work: it is recursive – components structure is auto-similar at any arbitrary level (hence the name 'Fractal'); it is completely reflexive, i.e., it provides introspection and inter-cession capabilities on components structure. These features allow for a uniform management of both the

so-called business and technical components (which is not the case in industrial component frameworks such as EJB [11] or CCM [12] which only deal with business components).

A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, which are under the control of the controller of the enclosing component. This allows for hierarchic components, in the sense that components may be nested at any arbitrary level. Fractal distinguishes *primitive* components (typically associated to a Java class implementing functional services) and *composite* components that only serve to build hierarchies of components, but without implementing themselves functional services.

A component can interact with its environment through *operations* at identified access points, called *interfaces*. As usual, interfaces can be of two sorts: *client* and *server*. A server interface can receive operation invocations (and return operation results of two-way operations), while a client interface emits operations. A *binding* is a connection between components, and more precisely between a client and a server interface. The Fractal model comprises bindings for composite and primitive components. Bindings on client ports of primitive components are typically implemented as language-level bindings (e.g. through type compatible variable affectations of interface references). The type of a binding might be a *collective* one or a *single* one (as default). In case of a collective one, a component may need, for achieving its functional work, to use (thus be bound to) a collection of components, instead of to a single component, all offering the needed interface.

A component controller embodies the control behavior associated with a particular component. Of importance is the following control: suspend (stop) and resume activities of the components in its content. Stopping then resuming is mandatory in order to dynamically change the binding between components or the inclusion of components. The important fact is that all such non-functional calls (stopping, resuming, binding, etc) propagates recursively to each internal component. This prevents the user manually triggering the same call on each sub-sub-...-sub component.

## 3 From Active Objects to Parallel, Distributed, and Hierarchical Components

### 3.1 Evaluation of the Needs

A component must be aware of parallelism and distribution as we aim at building a grid-enabled application by hierarchical composition; indeed, we need a glue to couple codes that probably are parallel and distributed codes as they require high performance computing resources. Thus components should be able to encompass more than one activity and be deployed on parallel and distributed infrastructures. Such requirements for a component are summarized by the concept we have named *Grid Component*.

Figure 1 summarizes the three different cases for the structure of a Grid component. For a composite built up as a collection of components providing common services, (fig. 1 c)) *collective communications* are essential, for ease of programming and efficiency purposes.

As general requirements, because we target high performance grid computing, it is very important to efficiently implement point-to-point and group method invocations, manage the deployment complexity of those components distributed all over the grid and possibly debug, monitor and reconfigure those running components – across the world.

### 3.2 ProActive Components

In the sequel, we describe the component framework we have designed and implemented using both Fractal and ProActive. It enables to couple parallel and distributed codes directly programmed using the Java ProActive library. A synthetic definition of what is a ProActive component is given below.

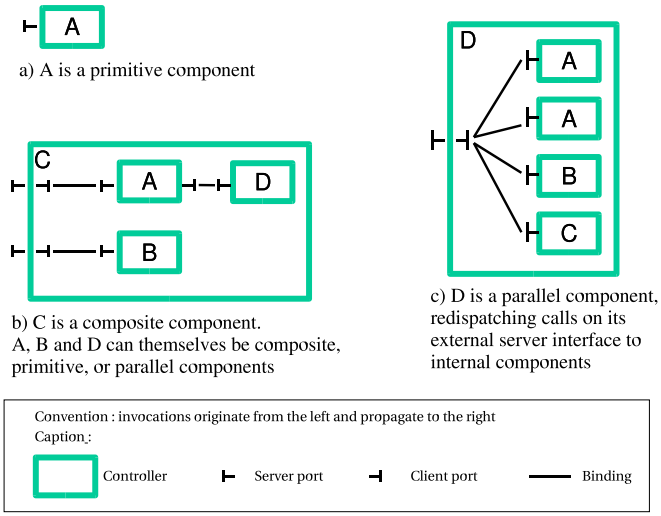



**Fig. 1.** The various basic architectures for a Grid component

**Definition of a ProActive component:**

- It is formed from one (or several) Active Objects, executing on one (or several) JVM
- It provides a set of server ports (Java Interfaces)
- It possibly defines a set of client ports (Java attributes if the component is primitive)
- It can be of three different types :
  1. primitive : defined with Java code implementing provided server interfaces, and specifying the mechanism of client bindings.
  2. composite : containing other components.
  3. parallel : also a composite, but redispatching calls to its external server interfaces towards its inner components.
- It communicates with other components through 1-to-1 or group communications.

**ProActive components can be configured using:**

- an XML descriptor (defining use/provide ports, containment and bindings in an Architecture Description Language style)
- the notion of virtual node, capturing the deployment capacities and needs

*Deployment of ProActive Components.* Components are a way to globally manipulate distributed and running activities, and in this context, obviously, the concept of *virtual node* is a very important abstraction. The additional need regarding the ones already solved by the deployment of active objects, is to be able to *compose virtual nodes*: a composite component is defined through a number of sub-components that already define their proper usage and mapping of virtual nodes. What should the mapping of the composite be ? For instance on fig. 2, when grouping two components in a new composite one, assume that each of the two sub-components, named respectively A and B, requires to be deployed respectively on VNa (further associated to 3 JVMs through the deployment descriptor) and the same for VNb (3 other JVMs). The question is how to define the mapping of the new composite ? Either distributed mapping is required (see fig. 2 a)) meaning that VNa and VNb must respectively launch different JVMs (a total of 6); or, a co-allocated mapping (see fig. 2 b)) where we try to co-locate as much as possible one activity acting on behalf of sub-component A and one activity acting on behalf of sub-component B within the composite C (on the example, only 3 JVMs need to be used).

Composition of virtual nodes is thus a mean to control the distribution of composite components.

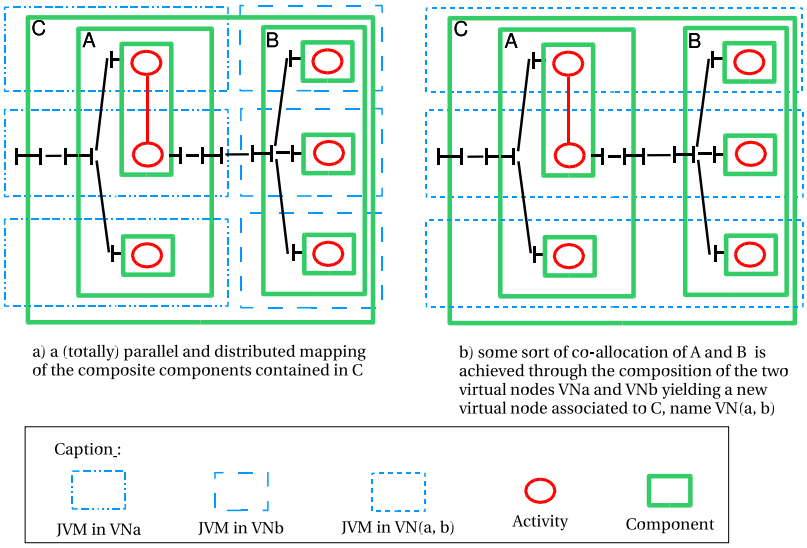



**Fig. 2.** Components versus Activities and JVMs

## 4 Implementation and Example

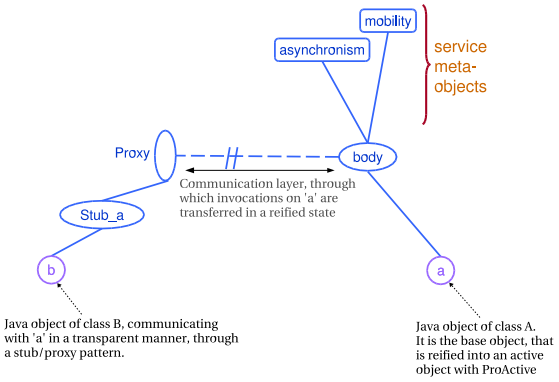
Fractal, along with the specification of a component model, also defines an API in Java. There is a reference implementation, called Julia, and we propose a new implementation, based on ProActive (thus providing all services offered by the Fractal library).

### 4.1 Meta-object Protocol

ProActive is based on a Meta-Object Protocol (MOP)(Figure 3), that allows to add many aspects on top of standard Java objects, such as asynchronism and mobility. Active objects are referenced through stubs, and the communication with them is done in the same manner, would they actually be remote or local.

The same idea is used to manage components: we just add a set of meta-objects in charge of the component aspect (Figure 4). Of course, the standard ProActive stub (that gives a representation of type A on the figure) is not used here, as we manipulate components. In Fractal, a reference on a component is of type `ComponentIdentity`, so we provide a new stub (that we call representative), of type `ComponentIdentity`, that references the actual component. All standard Fractal operations can then be performed on the component.

In our implementation, because we make use of the MOP's facilities, all components are constituted of one active object (at least), are they composite or primitive. Of course, if the component is a composite, and if it contains other




Java objects 'b' and 'a' can be in different virtual machines (the network being represented here between the proxy and the body, though the call might be local).Object 'b' has a reference on active object 'a' (of type `A`) through a stub (of type `A`) and a proxy. When 'b' invokes a method on 'stub_a',the call is forwarded though the communication layer (possibly through a network) to the body of the active object. At this point, the call can be intercepted by the meta-objects, possibly leading to some induced actions, and then the call is forwarded to the base object 'a'.

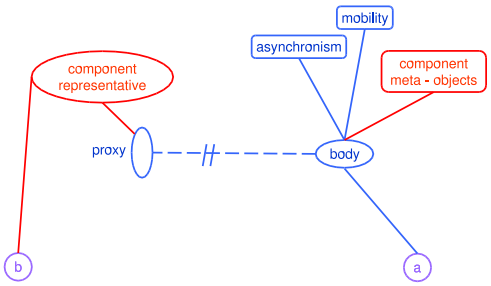
**Fig. 3.** ProActive's Meta-Object Protocol.

components, then we can say it is constituted of several active objects. Also, if the component is primitive, but the programmer of this component has put some code within it for creating new active objects, the component is again constituted of several active objects.

### 4.2 Integration within ProActive

To integrate the component management operations into the ProActive library, we just make use of the extensible architecture of the library. This way, components stay fully compatible with standard active objects and as such, inherit from the features active objects have: mobility, security, deployment, etc.

A particular point for the integration of Fractal and ProActive to succeed is the management of component requests besides functional requests. Reified method calls, when they arrive in the body, are directed towards the queue of requests. We assume FIFO is the processing policy. The processing of the requests in the queue is dependent on the nature of this request, and corresponds to the following algorithm :




A new set of meta-objects, managing the component aspect (constituting the controller of the component, in the Fractal terminology), is added to the active object 'a', and 'b' can manipulate the component based on 'a' through a specific stub, called *component representative*, of type `ComponentIdentity`. If the component is primitive, 'a' contains the functional code of the component.

**Fig. 4.** Component meta-objects and component representative.

```
loop
  if componentLifeCycle.isStarted()
      get next request
      // all requests are served
  else if componentLifeCycle.isStopped()
      get next component controller request
      // only component requests are served
  ;
  if gotten request is a comp. life cycle request
      if startFc --> set started = true ;
      if stopFc --> set started = false ;
  ;
;
```

Note that, in the stopped state, only controller requests are served. This means that a standard ProActive call, originating from a standard ProActive stub, will not be processed in the "stopped" state (but it will stay in the queue).

### 4.3 Collective Ports, Group Communications, and Parallel Components

The implementation of collective ports is based on the ProActive groups API (cf. [7]). According to the Fractal specification, this type of interfaces only has

sense on client interface, that would like to be bound to several server interfaces. Besides, one server interface can always be accessed by several client interfaces, the calls being processed sequentially. Specifying a server interface as "collective" wouldn't change its behavior.

The ProActive groups API allowing group communication in a transparent manner, the implementation of the collective interfaces slightly differs from the Fractal specification: instead of creating one new interface with an extended name for each member of the collection, we just use one interface (that is actually a group). Collective bindings are then performed transparently as if they were multiple sequential bindings on the same interface. Using a collective server interface will then imply using the ProActive group API formalism, including the possibility to choose between *scattering* and *broadcasting* of the calls [7]. A feature is that unbinding operations on a collective interface will result in the removal of all the bindings of the collection.

Furthermore, because we target largely distributed and parallel applications, we introduce a new type of component : *parallel components* (Figure 1 c)). These components are composite components, as they encapsulate other components. Their specificity relies in the behavior of their external server interfaces. These interfaces are connectable through a group proxy to the internal components' interfaces of the same type. This means that a call to the parallel component will be dispatched and forwarded to a set of internal components, that will process the requests in a parallel manner (see figure 5 a)).

### 4.4 Example

We present hereby an example of a component system built using our component model implementation.

Consider the following music diffusion system : a cd-player reads music files from a cd, and transmits them to a set of speakers situated in different rooms. Those speakers can convert music files into music we can listen to. They are incorporated in a parallel component, thus providing a single access interface to them (instead of connecting the cd player's output to each of the speakers).

Figure 6 gives an overview of the system, and represents the component model.

The system can be configured using the ADL (Architecture Description Language) that we provide for the components (Figure 7, coupled with the deployment descriptor, describing the physical infrastructure (Figure 8)).

When using the ADL, the configuration of the components is read from the descriptors, and the components are automatically instantiated, assembled and bound. Figure 9 shows an example of code used to manipulate the components, including instantiation, control and functional operations.

## 5 Related Work

We compare with closest related work in spirit, i.e. high-performance computing with composition of software components.

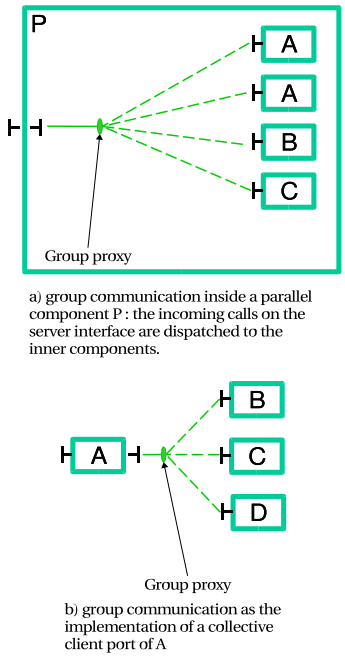



**Fig. 5.** Group communications allowing collective bindings and parallel components

*CCA.* The Common Component Architecture [1] is an initiative to define minimal specification for scientific components, targeting parallel and distributed infrastructures. Ideas are drawn from CCM for the sake of defining components by provide/use ports, calls/events through the usage of a Scientific IDL (SIDL). A toolkit of the CCA specification, called CCAT [2], provides a framework for applications defined by binding CCA-enabled components, in which all services (directory, registry, creation, connection, event) are themselves CCA components (wrapping external services). An instance of this framework, XCAT, permits to describe a component and its deployment using an XML document, which looks very similar to what we have also defined and implemented for ProActive components. In this XML-oriented implementation of CCA, the communication protocol used to implement the remote procedure call between a uses port method invocation and the connected provides port remote objects is based on SOAP. The main drawback of CCA is that the composition model is not related to any specific underlying distributed object oriented model so that the user lacks a clear and precise model of the composition (which is as important as having a clear and precise programming model).

*Corba Parallel Objects.* The Parallel Corba model [3] targets the coupling of objects whose execution model is parallel (in practice, a parallel object is incar-

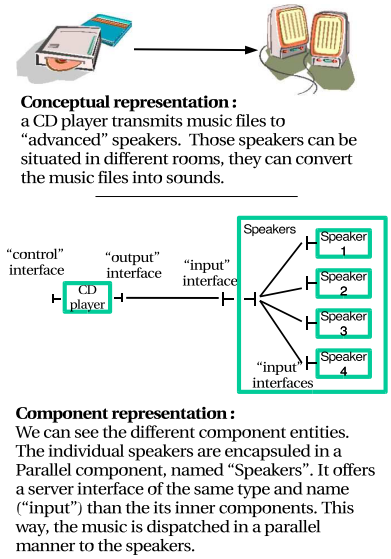



**Fig. 6.** A music diffusion system based on components

nated by a collection of sequential objects, and the execution model is SPMD; thus invoking a method on a parallel object invokes the corresponding method on all the objects of the collection, by scattering and redistributing arguments if needed). An implementation, PaCO++, achieves portability through the usage of standard CORBA IDL for object interactions. Notice that parallel and distribution issues are separated, as CORBA is only used to couple distributed codes, and parallel computations are usually managed with MPI. This is obviously an obstacle to easy grid computing.

*GridCCM.* GridCCM [4], a Parallel Corba component model, is a natural extension of PaCO++ motivated by the fact that a code coupling application can be seen as an assembly of components; however, most software component models (except PaCO++) only support sequential component. In order to have transparency in the assembly of components, a design choice was to make effective communications between parallel components be hidden to the application designer, by introducing collective ports that look like to be ordinary single-point ports. We propose the same sort of facility: ProActive components may also be built as parallel components by providing and using collective interfaces.

None of those approaches define hierarchical components as we have presented here. Moreover, we can encompass both parallel components in an SPMD style, or more generally parallel and distributed components following an MIMD execution model. We emphasize that we provide a unique infrastructure for functional and parallel calls and for component management, which is an alternative

```
COMPONENTS DESCRIPTOR
  Primitive–component  "cd–player"
      implementation = "CdPlayer"
      // name of the Java class with the functional code of the cd player
      VN = Node–player  //see deployment descriptor
      provides
          interface "control"
              signature = soundssystem.PlayerFacade
      requires
          interface "output"
              signature = soundssystem.Output
  Parallel–component  "speakers"
      VN = Node–speakers
      // the parallel component is just a facade to the real speakers
      provides
          interface "input"
              signature = soundssystem.Input
      contains
          primitive–component  "speaker"
              implementation = "Speaker"
              // functional code of the speaker
              VN = Node–speaker  (cyclic)  // see deployment descriptor
              // deployment descriptor will specify the location of
              // the instances (thus their number)
              provides
                  interface "input"
                      signature = soundssystem.Input
          Bindings
          // bindings to inner components are automatic for parallel
          // components between server interfaces of the same name
  Bindings
  // between client and server interfaces of the components
    bind "cd–player.output" to "speakers.input"
```

**Fig. 7.** Using the ADL to describe a component system (format is converted from XML)

to what is for instance done in GridCCM [4] (MPI, openMP, etc.,) for functional and parallel codes, and Corba for component management – binding, deployment, life-cycle management, ...).

## 6 Conclusion and Perspectives

We have successfully defined and implemented a component framework for ProActive, by applying the Fractal component model, mainly taking advantage of its hierarchical approach to component programming.

This defines a concept of what we have called *Grid components*. Grid components are formed of parallel and distributed active objects, features mobility, typed one-to-one or collective service invocations and a flexible deployment model. They also features flexibility and dynamicity at the component definition level.

We are working on the design of generic wrappers written in ProActive whose aim is to encaspulate legacy parallel code (usually Fortran-MPI or C-MPI codes).

We are also working on GUI-based tools to help the end-user to manipulate grid component based applications. Those tools will extend the IC2D monitor, which already helps in dynamically changing the deployment defined by deployment descriptors (cf. figure 8): acquire new JVMs, drag-and-drop active objects on the grid. We will provide *interactive* dynamic manipulation and monitoring

```
DEPLOYMENT DESCRIPTOR
VirtualNodes   // names of the virtual nodes
      VirtualNode name = "Node−player"
      VirtualNode name = "Node−speakers"
      VirtualNode name = "Node−speaker" − cyclic
      // cyclic: i.e. there will actually be several JVMs
Deployment   // what is behind the names of the virtual nodes
      mapping
      // correspondance between the names of the VNs and the JVMs
          Node−player −−> JVM1
          Node−speakers −−> JVM1
          Node−speaker −−> {JVM2, JVM3, JVM4}
          // 1 VN can be mapped onto a set of JVMs
      JVMs
          JVM1 created by process "linuxJVM"
          JVM2 created by process "rsh−computer1"
          JVM3 created by process "rsh−computer2"
          JVM4 created by process "globus−computer1"
Infrastructure
// how and where the JVMs specified above are created
    process−definition "linuxJVM"
    // this process creates a JVM on the current host
        JVMProcess class=JVMNodeProcess
    process−definition "rsh−computer1"
    // this process establishes an rsh connection
    // and starts a JVM on the remote host
    // (using the previously defined process "linuxJVM"
        rshProcess class=RSHProcess host="computer1"
        // computer1 could be in room1
        processReference = "linuxJVM"
    process−definition "rsh−computer2"
        rshProcess class=RSHProcess host="computer2"
        // computer2 could be in room2
        processReference = "linuxJVM"
    process−definition "globus−computer1"
        globusProcess class=GlobusGramProcess host="globus1"
        // globus1 could be in a room abroad
        processReference = "linuxJVM"
```

**Fig. 8.** Using the deployment descriptor to describe the physical infrastructure of a component system (format is converted from XML)

```
// CREATE THE COMPONENTS (for example speakers and cd_player)
ComponentIdentity speakers =
ProActive.newActiveComponent(speakers_parameters);
ComponentIdentity cd_player =
ProActive.newActiveComponent(cd_player_parameters);
// If the ADL is used, components instances can be retreived
// through the ComponentsLoader class :
// ComponentIdentity speakers =
// ComponentsLoader.getComponent("speakers");

// BIND THE COMPONENTS (Using the BindingController)
// (this is automatically done when using the ADL)
((BindingController)cd_player
.getFcInterface(BindingController.BINDING_CONTROLLER))
.bindFc(« output », speakers.getFcInterface(« input »);

// START THE LIFE CYCLE OF THE COMPONENTS
// (ENABLE THE COMPONENTS), using the LifeCycleController
((LifeCycleController)speakers
.getFcInterface(LifeCycleController.LIFECYCLE_CONTROLLER))
.startFc();
// this call is recursive, as the component contains other
// components (it also starts the inner components)

((LifeCycleController)cd_player
.getFcInterface(LifeCycleController.LIFECYCLE_CONTROLLER))
.startFc();


// INVOKE SOME ACTIONS ON FUNCTIONAL INTERFACES
// invoking a method of the Input interface
((Input)speakers
.getFcInterface(«input»))
.newMusic (music.mp3);

// invoking a method of the PlayerFacade interface
((PlayerFacade)cd_player
.getFcInterface(« control »))
.play();
```

**Fig. 9.** Using the API to manipulate components

of the components (besides what can be done by programming as exemplified by figure 9). For instance, it might be useful to generate an ADL such as the one on figure 7, and subsequently dynamically modify the description of the component application. Such tools could be integrated with computing portals and grid infrastructure middleware for resource brokering (ICENI [13], GridT [14], etc.), such as to build dedicated Problem Solving Environments [15].

We also investigate the following optimization: have functional method calls (either single or collective) bypass each inner composite component of a hierarchical component, so as to directly reach target primitive components – that are the only ones to serve functional service invocations. There is a non-trivial coherency problem to solve due to the concurrency of component management method calls (in particular, re-binding calls) towards encapsulating composite components.

**Acknowledgments.** This work was supported by the Incentive Concerted Action "GRID-RMI" (ACI GRID) of the French Ministry of Research and by the RNTL Arcad project funded by the French government.

## References

1. Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N.: Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. Cluster Computing **5** (2002)
2. Bramley, R., Chin, K., Gannon, D., Govindaraju, M., Mukhi, N., Temko, B., Yochuri, M.: A Component-Based Services Architecture for Building Distributed Applications. In: 9th IEEE International Symposium on High Performance Distributed Computing Conference. (2000)
3. Denis, A., Pérez, C., Priol, T.: Achieving portable and efficient parallel corba objects. Concurrency and Computation: Practice and Experience (2003) To appear.
4. Denis, A., Pérez, C., Priol, T., Ribes, A.: Padico: A component-based software infrastructure for grid computing. In: 17th International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, IEEE Computer Society (2003)
5. Caromel, D., Klauser, W., Vayssiere, J.: Towards seamless computing and metacomputing in java. Concurrency Practice and Experience **10** (1998) 1043–1061
6. Baude, F., Caromel, D., Huet, F., Mestre, L., Vayssière, J.: Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In: 11th IEEE International Symposium on High Performance Distributed Computing. (2002) 93–102
7. Baduel, L., Baude, F., Caromel, D.: Efficient, flexible, and typed group communications in java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, ACM Press (2002) 28–36 ISBN 1-58113-559-8.
8. Bruneton, E., Coupaye, T., Stefani, J.: Recursive and dynamic software composition with sharing. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02) (2002)
9. Fractal. (`http://fractal.objectweb.org`)

10. ProActive web site. (http://www.inria.fr/oasis/ProActive/)
11. Sun Microsystems: Enterprise Java Beans Specification 2.0 (1998) `http://java.sun.com/products/ejb/docs.html`.
12. OMG: Corba 3.0 new components chapter (2001) Document ptc/2001-11-03.
13. Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J.: ICENI: Optimisation of Component Applications within a Grid Environment. Parallel Computing **28** (2002)
14. Godakhale, A., Natarajan, B.: Composing and Deploying Grid Middleware Web Services Using Model Driven Architecture. In: CoopIS/DOA/ODBASE. Number 2519 in LNCS (2002) 633–649
15. Rice, J., Boisvert, R.: From Scientific Libraries to Problem-Solving Environments. IEEE Computational Science and Engineering (1996) 44–53

## 2.4 Administration

### 2.4.1 System and Network Management Itineraries for Mobile Agents.

E. Reuter and F. Baude. System and Network Management Itineraries for Mobile Agents. In *4th International Workshop on Mobile Agents for Telecommunications Applications, MATA*, number 2521 in LNCS, pages 227–238. Springer-Verlag, 2002.

# System and Network Management Itineraries for Mobile Agents

Emmanuel Reuter and Françoise Baude

OASIS Team, INRIA - CNRS - I3S
2004 route des Lucioles, BP 93
06902 Sophia Antipolis cedex – France
{First.Last}@sophia.inria.fr

**Abstract.** The technology of mobile agents has proven its usefulness for system and network management. In order to be effective, the integration of SNMP-based operations and the use of mobility should be effective. This paper presents a step forwards such an integration, by designing and implementing itineraries of mixed destination types: (1) a destination type that represents a location where a mobile agent migrates (for reaching a remote network for performance purposes, or/and for managing the location using a pure Java function); (2) a destination type that represents a location onto which the agent is not able to migrate, but for which an SNMP-based operation must be executed through a classical client-server interaction. The mixed usage of both kinds of destinations is unified within our solution, and as such greatly simplifies the programming of new system and network management operations.

## 1 Introduction

For several years now, the applicability and usefulness of mobile agent technologies for distributed System and Network Management (SNM) have been recognized. One of the main point is to delegate to autonomous and possibly mobile agents the administration tasks. As such, the network and computation loads are distributed instead of centralized towards and on the manager host [BPW98]. The Java programming language is today the most adequate for building such SNM platforms, as it provides: (1) a total portability on all kind of operating systems (due to the Java Virtual Machine), (2) built-in distribution and mobility management mechanisms (RMI – Remote Method Invocation–, dynamic class loading, serialization, etc.), (3) built-in security management mechanisms (permissions, security policies). Moreover, for this specific application domain, SNMP operations [Sta93] can be invoked from Java programs, in particular, using the AdventNet SNMP package [Adv98]. Several academic research platforms have been recently built in order to prove the effectiveness of Java mobile-agents based SNM: Mole [BHSR98], MAMAS [BCS99], MAP [PT00], JAMES [SRS99], just to mention a few.

A. Karmouch et al. (Eds.): MATA 2002, LNCS 2521, pp. 227–238, 2002.

One of the most tedious day-to-day task for a network and system manager is to keep the effective topology he/she has the responsability, in an up-to-date state, mainly in order to execute health monitoring. Fault diagnosis and network configuration are other important tasks which can also have some effect on the effective topology of the managed whole network. Using any mobile agent based platform for system and network management implies to first deploy the infrastructure (daemons in order to host mobile agents). Secondly, be able to program itineraries composed of system or network elements, Java-compliant or not, that mobile agents must visit in order to execute their management tasks.

The aim of our work is to introduce a transparent and easy-to-use mechanism such as to easily provide to mobile agents, up-to-date *itineraries* for executing their administrative task (on all or part of network elements of whatever type, Java or only SNMP compliant). Those itineraries will extend the common notion of what is an itineray for a mobile agent (i.e., a list of computers running a platform-specific daemon onto which a mobile agent will move to), in the following way: an itinerary for a SNM mobile agent will be considered as a list of elements to visit, not only locally after a move, but also remotely through a client-server interaction based on the classical SNMP protocol for instance. As such, we will provide a coherent, uniform and easy-to-use programming methodology that solves the difficult programming problem of mixing in a same mobile agent pure Java-based and pure SNMP-based management operations.

This programming facilities are made available into a system and management platform we are developing [RB02], using the *ProActive* library also developed in our team and supported by ObjectWeb (`www.objectweb.org`), a consortium for open source middleware . *ProActive* (`www.inria.fr/oasis/ProActive`) is a 100% Java library for parallel, distributed, concurrent computing with security and mobility, implemented on top of RMI as the transport layer. Besides remote method invocation services, *ProActive* features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronisation mechanisms, migration of active objects with pending calls and an automatic localisation mechanism to maintain connectivity for both "requests" and "replies". *ProActive* is an open libray, whose behaviour can be modified or extended by the classical inheritance mechanisms found in object-oriented languages. For the present work, it has been possible to extend the way a mobile agent follows its itinerary for the case when this itinerary is not only a list of destinations where to migrate.

In the following section, background on the model and usage of *ProActive* is given, followed by a quick overview of our SNM platform that is built upon this library for mobile code. Sections 3 and 4 detail how to build mixed itineraries and how their usage is already pre-programmed through the definition of a generic class. It remains only to programmers to extend this class in order to program a new specific mobile agent for a new management operation. Before concluding, we mention a few related works.

## 2 Background

### 2.1 The Model of Computation, Remote Invocation and Migration Featured by ProActive

As *ProActive* is built on top of standard Java APIs (Java RMI [Sun98b], the Reflection API [Sun98a],...). It does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified virtual machine. The model of distribution and activity of *ProActive* is part of a larger effort to improve simplicity and reuse in the programming of distributed and concurrent object systems [Car93,CBR96].

**Base Model.** A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls (see figure 1) sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [Car93]. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee (on Figure 1, step 1 blocks until step 2 has completed). The *ProActive* library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call, and which is put into the queue of pending requests in the same way as method calls.

**Mapping Active Objects to JVMs: Nodes.** Another extra service provided by *ProActive* (compared to Java RMI for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs, and to add a few services. *Nodes* provide those extra capabilities: a *Node* is an object defined in *ProActive* whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance: `rmi://lo.inria.fr/Node1`. As an active object is actually created on a *Node* we have instructions like: `a = (A) ProActive.newActive("A", params, "rmi://lo.inria.fr/Node1")`. If the active object must be created on the same Node as the one hosting the object initiating the creation, then,
`a = (A) ProActive.newActive("A", params)` is sufficient. Note that an active object can also be bound dynamically to a node as the result of a migration.

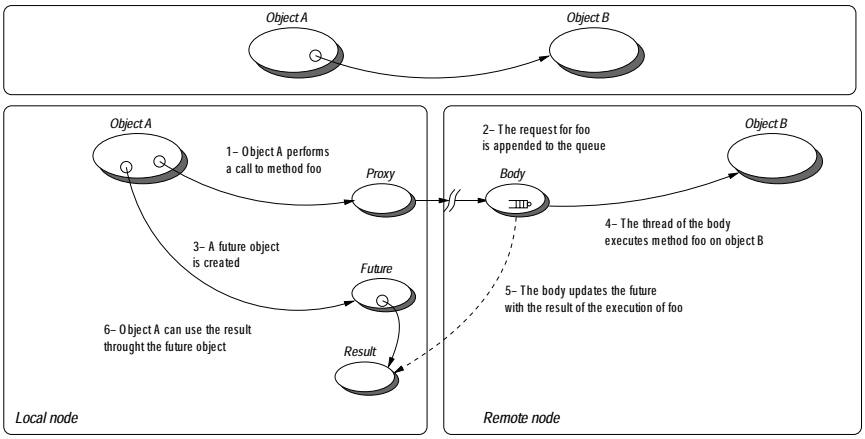



**Fig. 1.** Execution of a remote method call in *ProActive*.

**Migration of Active Objects from Nodes to Nodes.** The *ProActive* migration mechanism [BCHV00] follows a classical weak migration model. That is, the migration is triggered at specific points only, in the present case, when the next element in the requests queue of the active object is a `MigrateTo(...)` call. The migration involves deactivating and serializing the active object on the local node, and then deserializing and reactivating the active object on the remote one. More sophisticated abstractions have been built on top of the `migrateTo` primitive in order to control the migration pattern with a higher level of abstraction. An example of such an abstraction is `static void onArrival (String s)` which specifies as a string a method that will be executed upon arrival of the object at a node after its migration. Respectively, `onDeparture` allows us to declaratively specify code execution before migration. Finally, an *itinerary* abstraction allows us to specify a list of nodes to visit with various controls: `add, requestFirst, migrationFirst, itineraryStop`, `itineraryRestart`, etc. This is a `migrationStrategyManager`'s responsability to manage the travel of the active object according to its itinerary. For instance, if the `requestFirst` behaviour has been selected (it is the default one), then, in case the active object has pending remote method calls, then it will serve them before moving to the next location in its itinerary. Like all *ProActive* classes that configure the behaviour of an active object, the `migrationStrategyManager` can be extended such as to slightly modify the behaviour: it is exactly what we will done in order to build itineraries for System and Network Management.

### 2.2 Building and Maintaining the Effective Topology of the Network

The SNM platform we have designed and implemented [RB02] is able to dynamically discover all system or network elements that are reachable on the network.

To achieve this, one mobile *DiscoveryAgent* (itself programmed using *ProActive*) builds up a representation of the topology, by starting from an active equipment (a hub or switch for instance), and collecting and correlating SNMP variables of the various SNMP-enabled agents running on the current network. As such, it will discover all network elements, hosts and other devices and the way they are physically inter-connected. It gathers some information on each element, and registers all this in a specific server where this will be used for building itineraries for mobile agents as presented in the sequel. Such a server is called an *ItineraryServer*. This discovery process is then periodically re-executed in background, in order to have an up-to-date vision of the effective topology of the network. For each element that is discovered, the following kind of information is recorded in the corresponding ItineraryServer: the state (alive or not); network parameters (IP and Ethernet addresses); interface types; if this element executes an SNMP agent (and all its associated parameters, target agent SNMP, read-mode community name, UDP port number of the SNMP agent); if this element currently executes a ProActive Node, that is a specific daemon of our platform that could host a ProActive mobile object dedicated to a SNM task.

As the administrative domain of the managed enterprise or institution may be composed of several networks, the topology discovery can be launched and executed in parallel for each network and one *ItineraryServer* per network maintains the list of discovered elements on it. ItineraryServers are able to cooperate on demand in order to build itineraries that span several networks. Of course, the requirement is that elements belonging to a same network will appear close in an itinerary in order for a SNM mobile agent to avoid migrating more than once towards a given network.

The next sections will show how to take advantage of those ItineraryServers in order to easily program mobile agents that will execute their task by travelling according to itineraries.

## 3 Design of Itineraries for System and Network Management

### 3.1 Different Kinds of Destinations

The itinerary model for SNM we introduce is an extension of the basic itinerary concept provided by *ProActive* (a basic itinerary is a list of `NodeDestinations`, and for each, there is the ability to specify a method to execute on arrival). We have defined a new kind of destination, called an `SnmpDestination`, to which can also be associated a method to execute on arrival. An `SnmpDestination` collects all required information in order to execute a remote SNMP-based management function (see 2.2). Those various destination types are collected within a hierarchy (figure 2).

Simply extending the class `Destination` would enable to add a new kind of destination, such as for instance, one in order to support the version 3 of SNMP, or one to support the CMIP management [Sta93].

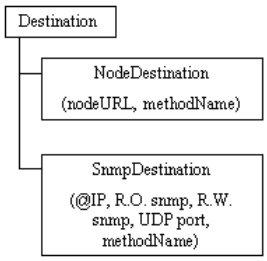



**Fig. 2.** Destination Class Hierarchy.

### 3.2 Itinerary Manager

The purpose of an `ItineraryManager` is to factorize all code that is needed such as to build up an itinerary according to the type of travel and administrative tasks the mobile agent will be launched for: for instance, a mobile agent itineray specific for visiting all SNMP-compliant network elements, or all printers.

All `ItineraryServers` will be asked to return the lists of elements or nodes they have collected during the topology discovery. Then, those lists will be filtered such as to obtain the required itinerary type. One particular itinerary type consists in visiting only a given sub-network. In this case, only the lists maintained by the ItineraryServer of this given sub-network will be requested. In order to provide a set of pre-defined itinerary types, we have defined an abstract `ItineraryManager` class from which we have extended several sub-classes, one for each common itinerary type. Of course, new itinerary patterns may be obtained by sub-classing `ItineraryManager`. During this sub-classing process, only the method to effectively filter the lists must be implemented by the programmer. Below are listed the different pre-defined itinerary types and their purpose:

- execute a complete management of the local network (i.e., it is assumed that the mobile agent is started on a *ProActive* node belonging to this network): `ItineraryManagerLocalNetwork` will return an itinerary collecting all SNMP-compliant elements in this local network;
  `ItineraryManagerLocalNodes` will return an itinerary collecting all *ProActive* nodes in this network, and it is assumed that all computers that must be visited must run a *ProActive* node such as to subsequently host the mobile agent,
- execute a remote management in a specific sub-network:
  `ItineraryManagerSubNetwork` will return an itinerary collecting all SNMP-compliant and *ProActive*-compliant elements; and the first element in the itinerary is one of the *ProActive* remote nodes such as to host the mobile agent for performance purposes (instead of remotely executing the management of the subnetwork);

– execute a remote management on all sub-networks: for obvious performance reasons, such an itineray will be built using `ItineraryManagerWholeNetwork`, as the result of concatenating itineraries for each sub-network, of the type `ItineraryManagerSubNetwork`. The purpose is that the mobile agent moves towards a remote sub-network only once.
  - `ItineraryManagerAllNetwork` is a restriction of the itinerary built with `ItineraryManagerWholeNetwork`, to SNMP-compliant elements only;
  - `ItineraryManagerAllNodes` is a restriction of the itinerary built with `ItineraryManagerWholeNetwork`, to *ProActive*-compliant elements only;
– execute a remote management on all sub-networks restricted to specific types of devices (for instance, restricted to routers, printers, computers, etc): `ItineraryManagerSpecificHosts`.

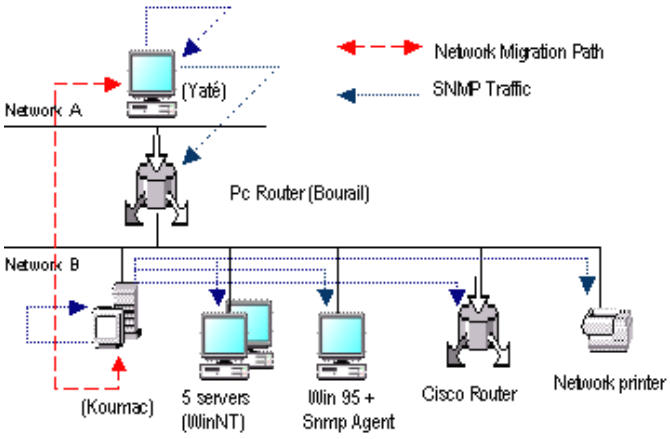



**Fig. 3.** Network diagram, showing the migration path and the SNMP traffic resulting from the itinerary built on table 1, such as to be manage all elements.

Figure 3 shows an example of a network that is built up with two sub-networks connecting elements that are only SNMP-compliant, Java-compliant or both. Suppose that we want to program a mobile agent such as to visit all those elements, then, using the right sub-class of ItineraryManager, an SNM itinerary such as the one shown in table 1 will be built. Notice for instance that it might be the case that an itinerary element (`Yate` for instance) acts as a recipient for the mobile agent but gets also managed using SNMP operations (in which case, two Destinations in the itinerary will be present).

**Table 1.** An Itinerary obtained with the `ItineraryManagerWholeNetwork` class for the network shown on figure 3.

| Type of Action | Element identification | OS Type | Type of Destination |
|---|---|---|---|
| Home NodeDestination | Yate | | |
| Migration | Koumac | Linux | NodeDestination |
| Client-Server | Koumac | UCD-SNMP | SnmpDestination |
| Client-Server | Server | WinNt | SnmpDestination |
| Client-Server | PC | Win95 | SnmpDestination |
| Client-Server | Server | WinNt | SnmpDestination |
| Client-Server | Network Printer | HP | SnmpDestination |
| Client-Server | Server | WinNt | SnmpDestination |
| Client-Server | Router | Cisco | SnmpDestination |
| Client-Server | Server | WinNt | SnmpDestination |
| Client-Server | Print Server | HP | SnmpDestination |
| Client-Server | Server | WinNt Term. Server | SnmpDestination |
| Client-Server | Server | Sco Openserver | SnmpDestination |
| Migration | Yate | Win XP | NodeDestination |
| Client-Server | Yate | | SnmpDestination |
| Client-Server | Bourail | Free BSD | SnmpDestination |
| End NodeDestination | Yate | | |

## 4 A Programming Model for a Mobile Agent Travelling According to a SNM Itinerary

The purpose is to provide to end-users a framework that may them help to easily develop new agents for system and network management, under the plaform we provide built upon *ProActive*. The idea is to mask as much as possible the heterogeneity of elements that must be managed and the travelling from elements to elements. Moreover, while an agent is following its itinerary, any other object must be able to remotely communicate with it if required. In this perspective, we provide the definition of a generic Java class, called `Agent` (see code on figure 4), from which active and mobile objects will be instanciated. As those objects are *ProActive* objects, they may execute pending remote method calls between considering any two `Destinations` in their itinerary. We will now detail those aspects.

*Automatic travel of the mobile agent following an itinerary.* As soon as the itineray is built up, the mobile agent will automatically be able to travel according to it. The specific `migrationStrategyManager` that we have tuned, in order to be in charge of itineraries mixing `NodeDestination` and `SnmpDestination`, will be in charge of that: when the next Destination is not a *ProActive* compliant element, then, no migration arises; each time the mobile agent reaches a `NodeDestination`, a method named `nodeOnArrival` gets automatically executed; each time it reaches a `SnmpDestination`, the method whose name is `snmpOnArrival` gets executed instead. This behaviour is automatically obtained

```
public class Agent implements java.io.Serializable {
  // the Destinations
  protected Destination dest;
  protected SnmpDestination snmpDest;
  protected NodeDestination nodeDest;
  private ItineraryManager itiManager; // my ItineraryManager
  private ArrayList visitedNodes; // collected visited nodes
  // constructor
  public Agent() {}
  public ArrayList getVisitedNodes() {
    return visitedNodes;
  }
  public void setCurrentDestination() {
    // automatically called by the migrationStrategyManager, in order to
    // set the Destination to the appropriate variable, e.g. NodeDestination
    // or SnmpDestination, and record the Destination visited
    dest = itiManager.getCurrentDestination();
    if (dest instanceof NodeDestination)
      nodeDest = (NodeDestination) dest;
    else snmpDest = (SnmpDestination) dest;
    visitedNodes.add(dest.toString());
  }
  public void nodeOnArrival(){};// what to do for a NodeDestination
  public void snmpOnArrival(){};// what to do for a SnmpDestination
  // Prepare the itinerary
  public void prepareItinerary(String itineraryServerHome,ItineraryManager itiManager) {
    // Record the ItineraryManager that must be used
    this.itiManager=itiManager;
    // ask it to build the required itinerary
    itiManager.prepareItinerary(itineraryServerHome);
  }
  // start to follow the itinerary
  public void startItinerary() {
    // start the travel
  }
  // add an urgent Destination to visit
  public void addUrgentDestination(Destination destination) {
    // the next Destination will be the new one
    itiManager.addUrgentDestination(destination);
  }
  // what to do at the end of the itinerary, if necessary
  public void atTheEnd(String methodName, Node homeNode) {
    itiManager.setEndMethod(methodName, homeNode);
  }
}// end of class Agent
```

**Fig. 4.** Generic Class Agent.

through the fact that the mobile agent extends the generic SNM agent class we provide, named `Agent` (see code on figure 4).

*Remote communication with a mobile agent.* As already mentioned, as mobile agents implement the programming model featured by *ProActive*, which generalizes the usual Java RMI mechanism, then, it is possible to remotely communicate with any travelling agent. For instance, it might be very useful to ask a mobile agent to take into account an urgent destination (by calling for instance the `addUrgentDestination(...)` method predefined by the `Agent` class. As explained in section 2.1, this method call will be executed before the next `Destination` in the itinerary gets visited.

```
public class AgentMix extends Agent implements java.io.Serializable {
  List AllMemInfo  = new ArrayList();
  List AllArplTable = new ArrayList();
  public AgentMix() {}
  public void nodeOnArrival() {  // implements Agent.nodeOnArrival
      SystemResources sr = new SystemResources();
      // Gets TOTAL MEM, USED MEM, FREE MEM into a String
      AllMemInfo.add(nodeDest.getURL()+sr.toString());
  }
  public void snmpOnArrival() {  // implements Agent.snmpOnArrival
      // gets the SNMP MIB Table ip.ipNetToMediaTable
      AllArplTable.addAll(new IpMacTable(snmpDest).getArpTable());
  }
  // Display the collected Data
  public void displayData() {
      System.out.println("Data collected :"+AllMemInfo+"\n"+AllArplTable);
  }
  // return "partial" results, in the sense that the agent may not have terminated its travel
  public List getPartialResult() {
      return AllMemInfo;
  }
}
```

**Fig. 5.** Scenario for an SNM agent extending `Agent`: it collects some Operating System Resources on each Java-compliant host and the ARP Table on each SNMP-compliant network element.

### Easy Programming of a New Mobile Agent

The only programming effort required for developing a new SNM agent (such as the one shown for instance on the code in figure 5) is to extend `Agent` and overrides the implementation of either the `nodeOnArrival` (in standard Java) or textttsnmpOnArrival methods (using the AdventNet SNMP package), or both, if the agent may execute both kinds of management operations. On the given example, the class `AgentMix` enables to collect some memory usage information for *ProActive*-compliant hosts (generally, computers), and to collect some variables stored in the SNMP MIB (here, `ip.ipNetToMediaTable` representing the MAC-IP table of the network element).

For launching a mobile agent executing this new management operation, then, the only requirement is to create this new active object, designate the required itineray type (by creating and passing an instance of the adequate `ItineraryManager` sub-class), and initiate the travel (see code on figure 6 for instance). Notice that the same agent may be reused and given an other kind of itineray, by just passing it an other instance of a `ItineraryManager` sub-class.

## 5 Related Work

In none of the SNM platforms we have studied, the idea of how to build an itinerary that will sustain the travel of a mobile agent has been explored so deeply as here. Sometimes, the itinerary is manually given by the graphical user interface the manager is sitting in front of, or, read in a text file. In other cases, such as in [OMDGG99], [GDM00], new elements are automatically recorded on

```
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.node.NodeFactory;
import mgt.agents.AgentMix;
public class Launch {
  public static void main(String args[]) {
    try {
     AgentMix mobileAgent = (AgentMix)ProActive.newActive("mgt.agents.AgentMix", null);
     mobileAgent.prepareItinerary(args[0], new mgt.itinerary.ItineraryManagerLocalNetwork());
     // NodeFactory.getDefaultNode is the ProActive Node onto the current JVM
     mobileAgent.atTheEnd("displayData", NodeFactory.getDefaultNode());
     mobileAgent.startItinerary();
     // do a job in parallel for instance
     List partialResult=mobileAgent.getPartialResult();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

**Fig. 6.** How to launch a mobile agent: for instance here, an agent of the AgentMix class that will manage the local network.

the management workstation, such as to automatically build accurate itineraries for SNM mobile agents. But, it is restricted to locations able to host mobile agents, so the problem of visiting non-Java elements is not addressed at all. To our knowledge, the idea of mixing and uniformely use different types of destinations that trigger different kinds of management methods, in the same itinerary, is new.

To our knowledge, only a few libraries for mobile agent programming introduce more structured itinerary patterns. For instance, the Ajanta library [TKA+02] features an itinerary description as a sequence of elements. Each element can either describe a single destination, or recursively a collection of destinations. The effective travel of a mobile agent will be structured in the same way.

## 6 Conclusion

We have introduced a new pattern for mobile agent itineraries, that proves to be particularily adapted to system and network management operations: indeed, in this kind of telecommunications application domain, the heterogeneity of elements is important, even with the wide acceptance and usage of the SNMP protocol. Our solution enables to mix several destinations into the same itinerary. If one would need to add an other kind of destination, such as CMIP for instance, only few classes should be extended or added: define a new adequate migrationStrategyManager, define adequate new sub-classes of Destination and ItineraryManager. Of course, the topology discovery mechanism provided by our SNM platform should also be extended such as to register this new type of element.

## References


Adv98. AdventNet. AdventNet SNMP tools. http://www.adventnet.net, 1998.

BCHV00. F. Baude, D. Caromel, F. Huet, and J. Vayssière. Communicating Mobile Active Objects in Java. *In proc. of the 8th International Conference - High Performance Computing Networking'2000 (HPCN Europe 2000), LNCS vol. 1823, pp 633–643*, 2000.

BCS99. P. Bellavista, A. Corradi, and C. Stefanelli. An Open Secure Mobile Agent Framework for Systems Management. *Journal of Network and Systems Management*, 7(3), 1999.

BHSR98. J. Baumann, F. Hohl, M. Straber, and K. Rothermel. Mole – Concepts of a Mobile Agent System. *World Wide Web 1, 3*, 1998.

BPW98. A. Bieszczad, B. Pagurek, and T. White. Mobile Agents for Network Management. *IEEE Communications Surveys 1, 1*, 1998.

Car93. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

CBR96. D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.

GDM00. Ghanbari M. Gavalas D., Greenwood D. and O'Mahony M. Advanced network monitoring applications based on mobile/intelligent agent technology. *Computer Communications Journal*, 23(8):720–730, April 2000. `http://citeseer.nj.nec.com/268291.html`.

OMDGG99. D. Greenwood O'Mahony M.J. D. Gavalas and M. Ghanbari. An infrastructure for distributed and dynamic network management based on mobile agent technology. *Proc. IEEE International Conference on Communications (ICC'99)*, 2:1362–1366, June 1999.

PT00. A. Puliafito and O. Tomarchio. Using Mobile Agents to implement flexible Network Management strategies. *Computer Communication Journal*, 23(8), April 2000.

RB02. E. Reuter and F. Baude. A mobile-agent and SNMP based management platform built with the Java ProActive library. *submitted*, 2002. `http://www.iufm.unice.fr/reuter/notes/submit02.ps` .

SRS99. P. Simões, R. Reis, and L. Silva. Enabling mobile agent technology for legacy network management frameworks. In *Proceedings of Softcom '99 Conference on Software in Telecommunications and Computer Networks*. IEEE Communications Society, October 1999.

Sta93. W. Stallings. *SNMP, SNMPv2, and CMIP*. Don Mills: Addison-Wesley, 1993.

Sun98a. Sun Microsystems. Java Core Reflection, 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/reflection/`.

Sun98b. Sun Microsystems. Java Remote Method Invocation Specification, October 1998. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.

TKA+02. A. Tripathi, N. Karnik, T. Ahmed, R. Singh, A. Prakash, V. Kakani, M. Vora, and M. Pathak. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140, May 2002.

### 2.4.2 A mobile-agent and SNMP based management platform built with the Java ProActive library.

E. Reuter and F. Baude. A mobile-agent and SNMP based management platform built with the Java ProActive library. In *IEEE Workshop on IP Operations and Management (IPOM 2002)*, pages 140–145, Dallas, 2002. ISBN 0-7803-7658-7

# A mobile-agent and SNMP based management platform built with the Java ProActive library

Emmanuel Reuter, Françoise Baude
Oasis, INRIA - CNRS - I3S
2004 route des Lucioles, BP 93
06902 Sophia Antipolis cedex – France
*First.Last*@inria.fr

## Abstract

*This paper presents our research into determining an adaptive and an up-to-date service developed for system and network management. By using a discovery process, the effective network topology is recorded and refreshed as necessary. In that way, by mixing collected information at each sub-network for example, an itinerary can be obtained that spans the whole administrative domain. Based on such possibility, we have developed a pre-programmed library that can be easily used in order to obtain a Mobile Agent for Network and System Management whose itinerary is dynamic.*

## I. Introduction

For several years now, the applicability and usefulness of mobile agent technologies for distributed System and Network Management (SNM) have been recognized. One of the main points is to delegate to autonomous and possibly mobile agents the administration tasks, as such, distributing the network and computation loads instead of centralizing them towards and on the manager host [1]. For a recent discussion of advantages of mobile agent based approaches in SNM, refer to [2].

The Java programming language is today the most adequate for building such SNM platforms, as it provides: (1) a total portability on all kind of operating systems (due to the Java Virtual Machine), (2) built-in distribution and mobility management mechanisms (RMI – Remote Method Invocation–, dynamic class loading, serialization, etc.), (3) built-in security management mechanisms (permissions, security policies). Moreover, for this specific application domain, SNMP operations can be invoked from Java programs, in particular, using the AdventNet SNMP package [3].

Several academic research platforms have been recently built in order to prove the effectiveness of Java mobile-agents based SNM: Mole [4], MAMAS [5], MAP [6], just to mention a few. They all have as a prerequisite the following: prior to the execution of any management operation, the network and system elements must run a daemon specific to the SNM platform, in order to be able to host a mobile agent. The daemon's role is to control the arrival and departure of mobile agents that come in order to execute their management operations locally, control their life-cycle and the multi-agent coordination. This of course requires that the system or the network element be Java-compliant to run this specific platform daemon. The management function is not mandatory pure JVM operations because it can be mixed with SNMP operations thanks to the Java/SNMP API.

However, in realistic infrastructures, network and system elements that the supervisor must manage are heterogeneous in the sense that not all of them are Java compliant (for instance, routers, printers are not currently able to execute JVMs); nevertheless, one can assume that they all run a standard SNMP agent, which can be remotely monitored through the SNMP protocol. Also in realistic infrastructures, the effective topology of the interconnected network and system elements is dynamic, as devices or computers may be up or down, devices or laptops may be added or removed, etc. Those elements may be part of different sub-networks (i.e., LANs), probably interconnected by higher latency and slower bandwidth links (i.e. WANs), as for instance in a multi-national or multi-regional enterprise.

One of the most tedious day-to-day task for a network and system manager is to keep the effective topology he/she has the responsibility, in an up-to-date state, mainly in order to execute health monitoring. Fault diagnosis and network configuration are other important tasks which can

also have some effect of the effective topology of the managed whole network.

Using any of the above mentioned mobile agent based platform implies to first deploy the infrastructure (daemons) and then to be able to tell a mobile agent which system or network elements it must visit in order to locally run the management function. As the topology may dynamically change and as some of the elements that must be managed can not be able to host a Java mobile agent, we think that those platforms lack some functionalities in order to be applicable for realistic infrastructures.

## II. Approach

In order to solve those real-world problems, our approach takes the form of a mobile agent based SNM platform which:

(1) automatically maintains an up-to-date effective topology of the network under management, which is composed of either SNMP compliant or Java-plus-SNMP compliant elements, structured into several sub-networks. In each sub-network, the communication bandwidth is assumed to be high enough such as to avoid using one mobile agent visiting each element in turn. Instead, one mobile agent could be moved on any one of the possible hosts running a platform specific daemon, and remotely execute the function on every element of the sub-network using the standard SNMP operations.

(2) provides to the mobile agent programmer, an API for building various *itineraries* [7] for mobile agents that reflect the up-to-date effective topology or part of it. Those itineraries are built up with two types of *destinations*: a destination type onto which the mobile agent can effectively move to[1] and then, if required, locally execute one pure Java or SNMP-based management function; an other "destination" type for which a SNMP-based management function will be remotely triggered, as it is not possible to host the mobile agent (either because the element is not Java-compliant, or it is not running the platform specific daemon). In this second type, the SNMP management function is remotely triggered by the mobile agent, from a destination of the first type it is currently located on.

Our SNM platform is built with ProActive, a 100% pure Java library for mobile and distributed computing based on active objects (`www.inria.fr/oasis/ProActive`) [8]. As ProActive's aim is to ease distributed programming (for instance, by abstracting away from synchronization and management of method invocations among remote active objects), extending the SNM platform with new management functions should be readily affordable to system and network managers and end-users of the platform.

[1]this is the usual sense of what is a destination in a mobile-agent itinerary

This paper will not specifically focus on the programming methodology, but instead, we will explain in section III how we build and maintain the effective topology, and in section IV, how itineraries are built in relation with the effective topology and transparently used by a mobile agent which "moves" from destination to destination in such itineraries. Section V studies some performance tradeoffs between pure SNMP remote management compared with this mixing of mobile agent and SNMP based management, on a real test bed. Section VI concludes while comparing with related works.

## III. Building and maintaining the effective topology of the network

The purpose is to dynamically discover all system or network elements that are reachable on the network, gather some information on each element, and register all this in a specific server where this will be used for building itineraries for mobile agents. Such a server is called an *ItineraryServer*. This discovery process is then periodically re-executed in background, in order to have an up-to-date vision of the effective topology of the network.

### A. Implementation

In the following, we consider a network as an IP subnet. Of course, the administrative domain of the managed enterprise may be composed of several networks,

A *DiscoveryAgent* programmed as a ProActive active and mobile object is in charge of the discovery of elements of a network, using only SNMP queries. The first and only element that needs to be queried in order to discover all other elements in the network is a piece of active equipment such as a seed router or a switch with a SNMP agent. Indeed, as such an active equipment systematically records all Ethernet addresses of the alive hosts on the network, it is enough to read and to correlate the corresponding SNMP MIB variables (e.g. `ip.ipNetToMediaTable`, `dot1dtBridge`) in order to build the topology of the network (the list of elements and the way they are interconnected). Nevertheless, it is necessary to filter those data (pairs IP/Ethernet addresses) such as to avoid to scan an IP subnet different as the current one. For each element that is discovered, the following kind of information is recorded in the *ItineraryServer* associated to each *DiscoveryAgent*: the state (alive or not); network parameters (IP and Ethernet addresses); interface types; if this element executes an SNMP agent; if this element currently executes a ProActive node, that is a specific daemon of our platform that could host a ProActive mobile object dedicated to a SNM task.

The *DiscoveryAgent* executing the discovery process must run on a ProActive node that may be local to the

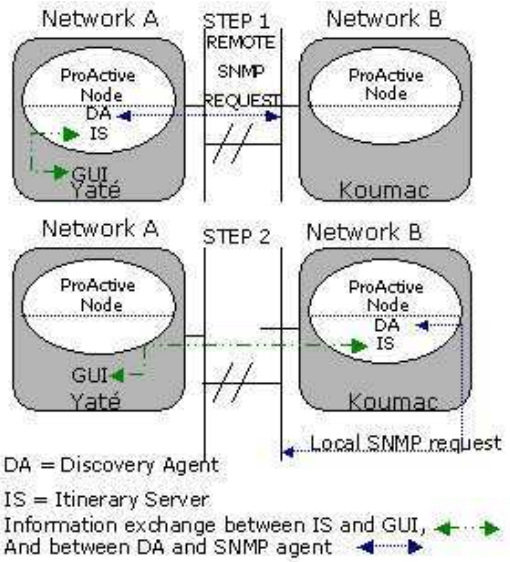



**Fig. 1. Discovery of the topology of network B, by remote SNMP investigation from network A (step 1), or by local SNMP investigation after the DiscoveryAgent was able to migrate from network A to network B (step 2).**

current network or not. For instance, this ProActive node may be the one which hosts the GUI (also a ProActive active object), but it is not mandatory. However, if during the discovery process of a remote network, the *DiscoveryAgent* locates a ProActive node, it migrates (and its associated *ItineraryServer* also) onto this node. As such, the discovery process of the network executes locally and so ends up faster (figure 1). On the contrary to some other SNM platforms, we do not need to make the assumption that prior to use[2] we already have at least one daemon specific to the platform running on each network.

## IV. Building and using itineraries for management tasks by mobile agents

### A. General idea and principles

An itinerary is a mixing of destinations onto which a SNM agent will effectively migrate and execute some Java or SNMP code on arrival, and of destinations that only represent elements for which the SNM task must take place without a move of the mobile agent (this requires the presence of an SNMP agent on those elements). Instructions for an SNMP agent will be triggered through a classical SNMP client-server interaction, originating from the mobile SNM agent, wherever the host it is actually located (it can be running on the same host, or it can be running on a host on the same network or even be located on a different network).

*ItineraryServer*s are able to cooperate on demand in order to build itineraries that span several networks. Of course, elements belonging to the same network will appear close in an itinerary in order for a SNM mobile agent to avoid migrating more than once towards a given network.

By requesting the up-to-date information recorded in the local *ItineraryServer* (which itself queries the others *ItineraryServer*s if required), any mobile agent can be provided with an itinerary that will enable to apply the SNM function to a set of elements, for instance:

- in an SNMP way only, inside the current network or among several networks (without any migration),
- in a mixed SNMP-Java way inside the same network (without any migration, except one in order to reach a ProActive node in the target network),
- in a mixed SNMP-Java way among several networks (with at least one migration for reaching every network in turn).

### B. Structure and usage of an itinerary

An itinerary is a list of *Destination*s, which can either be a *NodeDestination* or a *SNMPDestination*. Each destination must provide the following information: an identifier of the destination, and an identifier of a method name that must be executed on arrival. For instance: `<'''//koumac/node/''', ''echo''>` for a *NodeDestination*, `<''Bourail'', ''public'', ''snmpOnArrival''>` for a *SNMPDestination* (see Code in figure 2). As the SNM itineraries are built upon the basic ProActive mobile object itineraries, we were constrained by the fact that the method to execute on arrival can not contain any parameter. But, it is not very restrictive as while executing this method, the active object can locally trigger the execution of an other method with parameters (when an active object migrates, all its state is preserved).

An *ItineraryManager* is a class that provides some programming functions and as such serves as an interface between the SNM agent and the *ItineraryServer*. Upon creation, the agent provides some information regarding the type of itinerary it will need for visiting the system or network elements (e.g. MIX in the code in figure 2). Then, in order to effectively obtain the itinerary it must follow, it just calls one specific method defined in the *ItineraryManager* class (i.e. *setItinerary* which is in charge of querying `myLocalItineraryServer`), after that, it

[2]use in the broad sense, including the discovery process

only has to initiate the start of its ”visit” of all elements in the itinerary.

At any time, thanks to a method call originating for instance from an other mobile agent or from the GUI running on the host manager, the SNM agent (its *Itinerary-Manager*) may be told to insert into its itinerary a new *Destination*, possibly in front of the itinerary. This is an easy way of forcing an agent to go back home for instance, or to urgently manage an element.

```
public class MyAgent implements java.io.serializable {
  // Triggered for a NodeDestination
  public void echo() {
    System.out.println("MyAgent.echo()");
  }
  // Triggered for a SNMPDestination
  public void snmpOnArrival() {
    // Gets  SNMP parameters from ItineraryManager
    SNMPDestination snmpDest =   (SNMPDestination)
       itiManager.getCurrentDestination();
      // do your SNM job !
  }
 // Prepare and start to follow an Itinerary
 public void start(String myLocalItineraryServer) {
    // Create an ItineraryManager
    itiManager = new ItineraryManager(MIX);
    // Set the home for locating our local
    // network ItineraryServer
    // and prepare for our test an itinerary
    // in order to 'visit' all the networks
    itiManager.setItinerary(myLocalItineraryServer);
    // Ask the ItineraryManager to start the migration
    itiManager.startItinerary();
  }
 public static void main(String args[]) {
    try {
      // Create an Active Object
      MyAgent myAgent =   (MyAgent)
       ProActive.newActive("mgt.agents.MyAgent", null);
      // prepare the itinerary and go !
      myAgent.start(args[0]);
     } catch (Exception e) { e.printStackTrace(); }
  } // main
}// end of class MyAgent
```

**Fig. 2. Code Example of a mobile agent and a transparent itinerary usage**

## V. Performance Evaluation

As itineraries built with our platform can mix remote SNMP management or local SNMP management after a migration on the host, we have studied tradeoffs of such mixing. The tradeoffs depend on the bandwidth of the links that connect networks (in our case, 2 networks), on the size of the mobile agent when it must cross this link in order to reach some other network, on the size of the data collected by the SNMP management function (either a fix number of SNMP variables, or a variable number as when collecting a routing table for instance). Moreover, the purpose of those evaluations is to check that our SNM platform behaves correctly and yields reasonable performances for realistic infrastructures.

### A. Benchmark Configuration

In our test-bed, there are two networks (see figure 3). In network A, two computers (`Yate` and `Bourail`) and in the other, eleven computers of different power and capabilities. Those machines are PCs (running Win95, Linux, WinNT, Sco OpenServer, WinNT Terminal Server), Network Printers (HP 4100 and HP 2100) all executing an SNMP agent. In order to make the network bandwidth between the two local LANs varies, we have used a Pentium at 133Mhz (`Bourail`) running a Free BSD operating system with `ip_dummynet` (a bandwidth limiter) [9] (fig 3). As such, we could simulate a 50Kbps up to a 10Mbps link connecting both LANs. Each network is a 100Mbps switched Ethernet LAN.

In order to simulate a bigger configuration, longer itineraries are obtained by increasing the number of round trips. In the case of the itinerary using mobility, each round trip is as follows: one migration at the start to go from the workstation `Yate` to `Koumac` and one at the end to go back to the initial ProActive node on `Yate`. Notice that we do not empty the agent when it comes back on `Yate`, because the purpose is to simulate an itinerary that spans a whole administrative domain compound of several networks interconnected by low bandwidth links. As such, its size will grow. Of course, previous work has already pointed out a possible improvement: empty (or temporarily store) the data the mobile agent is carrying out with it before migrating again [10]. Alternatively, provide an itinerary whose pattern is a star-shape route [11], where the mobile agent migrates back and forth between the central node and the other nodes, just to deliver its results[3]. In our framework, we also might program a remote method call between the agent and the source node, such as to transmit the results before migrating to the next network. However, it is not the purpose here to evaluate those optimizations or alternative traveling patterns. In the case of the itinerary not using mobility, each round trip is as follows: each element mentioned in the itinerary is a *SNMPDestination*, and as such, all SNMP read operations have to be executed from `Yate`, whatever be the number of networks.

### B. Comparison between remote SNMP function and mobile agent plus local SNMP function

The first SNMP function we have programmed is to read onto each element mentioned in the itinerary, a fixed number of SNMP variables that lie in the System MIB-II branch (e.g. system.sysDescr).

When the link bandwidth is equal to 50Kbps, as when for instance the host manager is running on a laptop

[3]Such an itinerary type can easily be provided by extending the *ItineraryManager* class

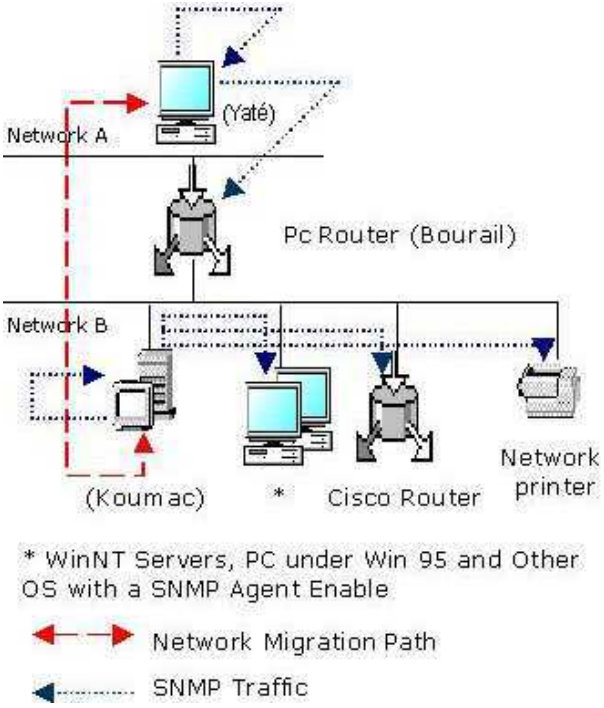



**Fig. 3. Network diagram**

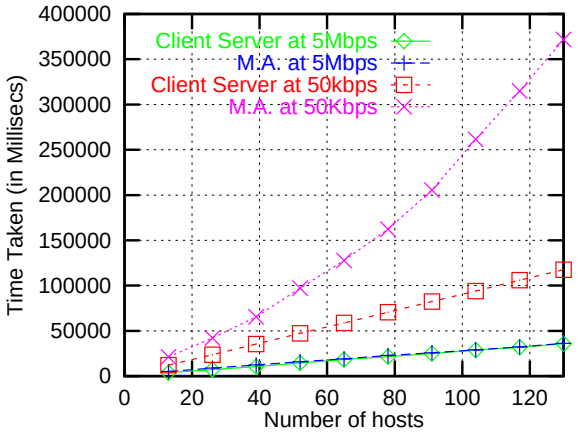



**Fig. 4. Retrieval of a small number of SNMP variables per element**

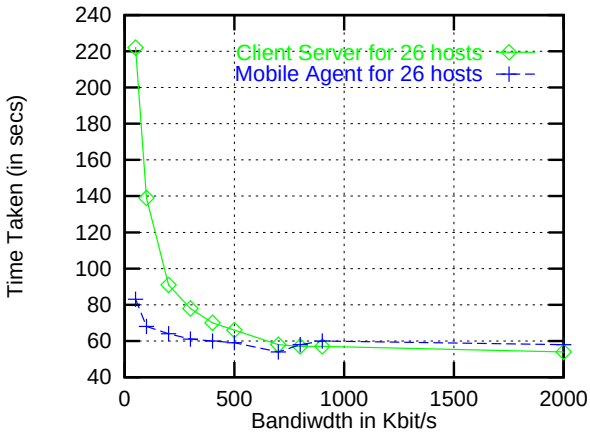



**Fig. 5. 26 Destinations in the itinerary; retrieval of more than 7 SNMP variables for each**

connected to the network with such a very low bandwidth connection, figure 4 proves that classical SNMP monitoring shows better performances. Indeed, in the mobile agent based experiment, the performance is constrained by the migration performance: the mobile agent migrates twice at each round trip and its size grows with the number of visited hosts. As such, its migration is especially time consuming on a low bandwidth link. The same behaviour can be observed as when the link bandwidth is equal to 100Kbps. However, as soon as the link bandwidth exceeds 1Mbps (as for 5Mbps as shown on figure 4) both kinds of experiments have the same duration: the growing size of the agent is no more a bottleneck for the usage of mobility in the itinerary. If the total amount of information read in the SNMP MIBs increases (e.g. ip.ipRouteTable and ip.ipNetToMediaTable), then the link bandwidth might be the limiting factor even for the pure SNMP-based experiment (see figure 5): in this case, the experiment does not not take advantage of the effect of proximity (proximity means that the read operations in the MIBs of network elements are triggered locally from a host located in the same network instead of remotely, from the initial host for instance).

## VI. Discussion and conclusions

Distributed SNM platforms using mobile agents are useful and even efficient in a wide range of hardware configurations, especially when the links interconnecting networks are of low bandwidth, and the output data size of SNM tasks are huge: it is then possible to aggregate and maybe compress and filter those data at the place or in the same network where they have been collected, before sending them back to the manager host. Such considerations and conlusions had already been made for instance in [12], [13], [14] but we had to check that the same behaviour occurs within our SNM platform. Many models have been defined in [14] in order to characterize

SNM applications: the itinerary pattern mixing migration and SNMP retrieval of information we have introduced could correspond to a new model combining the Static Centralized (SC) and Migratory (MG) ones, taking advantage of proximity and locality. Nevertheless, we have not yet studied the usage and effect of distribution in our SNM platform, as evaluated in [14]. However, as the ProActive library does provide the notion of groups of mobile active objects [15], it should be feasible to create a group and to dynamically provide to each peer one itinerary that spans a given part of the network. Then, each peer would independently follow its own itinerary, and peers might also be able to communicate in order to aggregate, correlate and exchange the collected data. This pattern of SNM can be modeled by an extension of the Static Delegated (SD) model [14], in which an agent collecting information is not static but can migrate towards an other network or towards the manager host, or even within its assigned network if necessary.

Another important focus of our SNM platform is the ease of deployment of the support infrastructure for mobile agents. Indeed, it is very constraining if the requirement is to install and run a mobile agent support for every managed element prior executing any SNM task[4]. Thanks to the itinerary pattern we have introduced, it is not mandatory to meet this requirement. As the dynamic discovery of the topology of the administrative domain executes, running ProActive nodes will be located and registered into *ItineraryServer*s so as to subsequently be included as *NodeDestination*s into mobile agent itineraries. However, if no ProActive node is found on a given network, then, all its elements can still be managed using a classical SC model.

Arguments for isolating the behavioral logic part from the itinerary part of mobile agents, as done here, include the one mentioned in [11]: building an efficient itinerary customized for each different network is a time-consuming and difficult operation without any knowledge of the network. We can add the following argument: traveling along an itinerary that reflects an up-to-date topology is impossible if the itinerary is statically embedded in the agent at programming time or at the time of the SNM platform deployment. Solutions to both problems rely on mobile agents dynamically retrieving their itinerary from some predefined entities (*AgentPool*s and *NavigatorAgent*s in [11], *ItineraryServer*s in the present work), whose task is to manage and combine information about the network. It would be possible to implement the first of the above mentioned arguments by adequate computations within *ItineraryServer*s at the end of each new topology discovery process execution.

[4]However, we must assume that every managed element runs an SNMP agent, if it does not run a ProActive node

## References


[1] A. Bieszczad, B. Pagurek, and T. White., "Mobile Agents for Network Management," *IEEE Communications Surveys 1, 1*, 1998.

[2] G. M. Gavalas D., Greenwood D. and O. M, "Advanced network monitoring applications based on obile/intelligent agent technology," *Computer Communications Journal*, vol. 23, no. 8, pp. 720–730, April 2000.

[3] "AdventNet SNMP tools," http://www.adventnet.net, 1998.

[4] J. Baumann, F. Hohl, M. Straber, and K. Rothermel, "Mole – Concepts of a Mobile Agent System," *World Wide Web 1, 3*, 1998.

[5] P. Bellavista, A. Corradi, and C. Stefanelli, "An Open Secure Mobile Agent Framework for Systems Management," *Journal of Network and Systems Management*, vol. 7, no. 3, 1999.

[6] A. Puliafi to and O. Tomarchio, "Using Mobile Agents to implement flexible Network Management strategies," *Computer Communication Journal*, vol. 23, no. 8, 2000.

[7] E. Reuter and F. Baude, "System and network management itineraries for mobile agents," in *4th International Workshop on Mobile Agents for Telecommunications Applications, MATA*, 2002.

[8] D. Caromel, W. Klauser, and J. Vayssière, "Towards Seamless Computing and Metacomputing in Java," *pp. 1043–1061 in Concurrency Practice and Experience, 10(11–13)*, 1998.

[9] L. Rizzo, "Ip dummynet," Dip. di Ingegneria dell'Informazione, Univ. di Pisa, http://info.iet.unipi.it/˜luigi/ipdummynet.

[10] Y. Aridor and D. Lange, "Agent Design Patterns: Elements of Agent Application Design," in *Second International Conference on Autonomous Agents (Agents'98)*. ACM Press, pp. 108–115.

[11] I. Satoh, "A Framework for Building Reusable Mobile Agents for Network Management," in *Network Operations and Managements Symposium (NOMS'2002)*. IEEE Communication Society.

[12] M. Rubinstein and O. Duarte., "Evaluating tradeoffs of mobile agents in network management," *Networking and Information Systems Journal*, vol. 2, 1999.

[13] A. Sahai and C. Morin, *Software Agents for Future Communications Systems*. A.L.G.Hayzelden and J. Bigham (Eds), Springer Verlag, 1999, ch. Mobile Agents for Managing Networks : MAGENTA perspective.

[14] P. Simões, J. Rogrigues, L. Silva, and F. Boavida, "Distributed Retrieval of Management Information: Is It About Mobility, Locality or Distribution ?" in *Network Operations and Managements Symposium (NOMS'2002)*. IEEE Communication Society.

[15] L. Baduel, F. Baude, and D. Caromel, "Effi cient, Flexible, and Typed Group Communications in Java," in *Joint ACM Java Grande - ISCOPE 2002 Conference*.

# Troisième partie

# Annexes

# Chapitre 1

# Curriculum Vitae


**Françoise Baude**

Maître de Conférences
Université de Nice Sophia Antipolis - I3S - C.N.R.S. UMR 6070

INRIA Sophia-Antipolis
2004 route des Lucioles, B.P. 93, F-06902 Sophia Antipolis Cedex
04 92 38 76 71, Fax. : 04 92 38 76 44
baude@unice.fr
`http ://www.inria.fr/oasis/Francoise.Baude/`


## 1.1 État civil

Nom : Baude épouse Dreysse
Prénom : Françoise
Née le : 30 Mars 1966 à Laxou (54)
Nationalité : Française
Situation de famille : mariée, 2 enfants, nés en 1996 et 1997.

## 1.2 Formation et diplômes

– Licence - Maîtrise Informatique (Juin 1987), Université des Sciences et Techniques du Languedoc (USTL), Montpellier

- DEA Informatique option *bases de données et systèmes distribués* (Juin 1988), mention AB, USTL Montpellier
  Sujet de recherche : Contrôle de concurrence dans les systèmes distribués.
- Doctorat en Sciences de l'Université de Paris-Sud (Décembre 1991), "Utilisation du paradigme acteur pour le calcul parallèle",
  Président : J.-P. Sansonnet, Rapporteurs : G. Agha, M. Habib, Examinateur : J.-P. Briot, Directeur : G. Vidal-Naquet.

## 1.3 Activités professionnelles

- Octobre 88 à Décembre 91 : Doctorante avec une bourse CIFRE au sein d'Alcatel-Alsthom Recherche, Marcoussis
- Décembre 91 à Décembre 92 : Post-doctorat financé par une bourse INRIA
  Department of Computer Science, University of Warwick, Coventry UK (décembre 91 à juin 92)
  Department of Computing and Information Science, Queen's University, Kingston Canada (juin à décembre 92)
- Octobre 92 à Septembre 93 : ATER à l'Université de Paris-Sud (Faculté des Sciences, Département Informatique), recherches menées au sein du laboratoire LRI (URA CNRS 410).
- Depuis Octobre 93 : Maître de Conférences à l'Université de Nice - Sophia Antipolis (Faculté des Sciences, Département Informatique), recherches menées au sein du laboratoire CNRS I3S (UMR 6070).
- D'Octobre 95 à Décembre 98 : membre du projet SLOOP (Simulation, Langages à Objets et Parallélisme), projet commun Université de Nice Sophia Antipolis/CNRS I3S/INRIA Sophia Antipolis.
- Depuis Janvier 1999 : membre du projet OASIS (Objets Actifs, Sémantique, Internet et Sécurité), projet commun Université de Nice Sophia Antipolis/CNRS I3S/INRIA Sophia Antipolis.
- En délégation à l'INRIA Sophia-Antipolis, dans l'équipe OASIS, de Septembre 2004 à Septembre 2006.
- Titulaire d'une Prime d'Encadrement Doctoral et de Recherche depuis Octobre 2001, renouvelée en Octobre 2005.

## 1.4 Conseils et commissions

- Membre élu de la Commission de Spécialistes de la 27ème section de l'Université de Nice - Sophia Antipolis, de Mars 1995 à Mars 1998, puis à nouveau depuis Mars 2000.
- Représentante suppléante puis permanente (depuis janvier 2005) de l'équipe OASIS au comité des projets de l'I3S.
- Membre du comité de pilotage du pôle GSP (Grille, Système et Parallélisme) du GDR (Groupement De Recherche) ASR (Architectures, Systèmes, Réseaux) du CNRS, à partir de 2006

## 1.5 Activités d'Enseignement

### 1.5.1 Participation aux charges administratives liées à l'enseignement

- Depuis septembre 2000 et jusqu'en septembre 2004, coordinatrice de la licence d'informatique de l'UNSA (environ 80 étudiants par an).
- De septembre 2002 à septembre 2003, coordinatrice de la maitrise d'informatique de l'UNSA (65 étudiants), en remplacement "au pied levé" du coordinateur initialement prévu.
- Pour l'année universitaire 1998-1999, responsable de la répartition des enseignements entre les différents vacataires intervenant au sein du département d'informatique de la faculté des sciences de l'UNSA.

Coordonner des filières nécessite entre autre chose :
- la participation à la définition des maquettes, de manière consensuelle avec les équipes pédagogiques,
- l'organisation du calendrier et des emplois du temps,
- la préparation des jurys et leur conduite,
- et surtout, le suivi personnalisé des étudiants (notamment des étudiants en difficulté, ou n'ayant pas un parcours classique).

### 1.5.2 Enseignements dispensés

Depuis ma nomination sur un poste de maître de conférences à l'UNSA en octobre 1993, j'ai eu à mettre en place un certain nombre d'enseignements

en premier, second et troisième cycles (rédaction du cours, des sujets de TDs et de TPs, sujets d'examen et de projets le cas échéant, encadrement des équipes pédagogiques, etc). Depuis ma délégation à l'INRIA obtenue en septembre 2004, pour 2 ans, j'ai continué à enseigner en troisième cycle Master Recherche.
Le tableau ci-dessous présente de manière synthétique les différents enseignements que j'ai dirigés et dispensés.

| | | | |
|---|---|---|---|
| 1*er* cycle | *Internet et Bureautique* | 20h TP | Tronc-commun deugs MIAS-SM-MASS première année (800 étudiants) |
| | *Outils Formels pour l'Informatique* | 28h C-28h TD | Deug MIAS, mention Mathématique-Informatique 2ème année |
| | *Systèmes Informatiques* | 28h C-14h TP | Deug MIAS, mention Mathématique-Informatique 2ème année |
| | *Concepts des systèmes d'exploitation* | 12h C-12h TD | IUP MIAGE 1ère année |
| 2*eme* cycle | *Gestion des ressources et de la concurrence* | 16h C-16h TD-8h TP | Licence Informatique |
| | *Utilisation avancée des systèmes d'exploitation* | 10h C-8h TP | Licence Informatique |
| | *Systèmes distribués* | 12h C | IUP MIAGE 3ème année |
| 3*eme* cycle | *Environnements et Algorithmique Parallèle et Distribuée* | 15hC-8h TP | ESSI 3ème année, filière Systèmes et Architectures Réparties |
| | *Utilisation et principes des systèmes d'exploitation* | 9h C-12h TP | Mise à niveau - DESS Télécommunications |
| | *Algorithmique Distribuée* | 6h C | Partie d'un module de tronc commun du DEA/ Master STIC spécialité Recherche *Réseaux et Systèmes Distribués* |

| | | | |
|---|---|---|---|
| | *Programmation parallèle fonctionnelle* | 4h C | Partie d'un module d'option commun aux DEA / Master STIC spécialité Recherche *Programmation : Modèles, Langages, Techniques* et au Master STIC Recherche *Réseaux et Systèmes Distribués* |

Par ailleurs, j'ai pris part aux travaux pratiques d'un certain nombre d'autres enseignements.

| | | | |
|---|---|---|---|
| 1*er* cycle | *Bureautique* | 14h TP | Deug MIAS, mention Mathématique-Informatique 1ère année |
| 2*eme* cycle | *Programmation orientée objet en Java* | 26h TP | Licence Informatique |
| | *Outil de rédaction LaTeX* | 6h TP | Licence Informatique |
| | *Compilation-Technologies XML* | 21h TP | IUP MIAGE 2ème année |

## 1.6 Activités de recherche

### 1.6.1 Organisation et évaluation de la recherche

**Présidence de comité**

- Présidente du comité d'organisation du colloque STRATAGEM'96 qui s'est déroulé au centre INRIA de Sophia-Antipolis, du 8 au 10 juillet 1996. Son objectif était de présenter à la communauté internationale les principaux résultats obtenus dans le cadre du projet CNRS Stratagème.
- Présidente du comité de lecture d'un numéro spécial de Calculateurs Paralléles, édité chez Hermès, sur le *Metacomputing*, finalement, paru sous forme d'un ouvrage collectif [E2][1]
- Présidente du comité de programme de RenPar'15 (Rencontres Francophones du Parallélisme)

[1]Se référer à la liste des publications fournie en chapitre 2 de l'annexe

- Présidente du comité d'organisation de RenPar/SympAAA/CFSE, *Rencontres Francophones en Parallélisme, Architecture, Adéquation Algorithmes Architecture et Système*, octobre 2003, La Colle sur Loup. Présidente de la session "Invité" de RenPar'15.
- Présidente du comité de lecture du numéro spécial de la revue Techniques et Sciences Informatiques, sur les recherches actuelles en parallélisme, suite à une sélection des meilleures communications faites lors de RenPar'15, voir [E4].

**Participation à des comités d'organisation**

- Membre du comité d'organisation de la conférence ECOOP (European Conference on Object-Oriented Programming), qui s'est tenue à Sophia-Antipolis et Cannes en juin 2000. J'étais co-chair pour les tables rondes, puis éditrice de l'une d'entre elles [E1].
- Vice-chair du topic " Object Oriented Architectures, Tools and Applications" de la conférence EuroPar, Münich, Septembre 2000.
- *Local vice-chair* pour l'organisation de la conférence internationale IEEE IPDPS (International Parallel and Distributed Processing Symposium), qui s'est tenue à Nice en avril 2003.
- Participation à l'organisation du Premier Grid PlugTests, organisé conjointement entre l'équipe OASIS et l'ETSI, 18-20 octobre 2004, Sophia-Antipolis.
- Participation à l'organisation de la conférence Grids@work, 10-14 octobre 2005, Sophia-Antipolis, organisée conjointement entre l'équipe OASIS et l'ETSI. Montage et suivi d'un contrat de collaboration entre l'ETSI et l'INRIA (notamment concernant la mise en place d'une grille de calcul pour le 2nd Grid Plugtests and contest) d'avril à décembre 2005 ; présentation de l'événement au *European Grid Technology Days 2005* organisés par l'IST Unité Grid, Bruxelles, Mai 2005 ; recherche de sponsors et de participants de type industriel ou académique ; présidence de la session "Industrielle" ; élaboration de communiqués de presse (voir notamment la rubrique Chronique de [E4]).
- *Publicity chair* pour l'Europe concernant la 15ème édition de la conférence internationale IEEE HPDC (High Performance Distributed Computing), Paris, juin 2006
- Participation à l'organisation de la conférence Grids@work II, 27 nov.-1er décembre 2006, Sophia-Antipolis, organisée conjointement entre

l'équipe OASIS, l'ETSI et CoreGRID.

### Participation à des comités d'édition

- Membre permanent du comité de rédaction de Calculateurs Parallèles, revue AFCET-LBI, éditée chez Hermès jusqu'en 2003, de Mars 1996 à Décembre 2001.
- Membre du comité de lecture pour la revue TSI (Techniques et Sciences Informatiques), sur le thème "Systèmes à composants adaptables et extensibles", décembre 2002.
- Membre du comité de lecture pour la revue *Concurrency and Computation : Practice and Experience*, sur un numéro spécial autour des intergiciels pour la grille, Mars 2006.

### Participation à des comités de programmes

**Internationaux :**

- Membre du comité de programme du workshop "Infrastructure and Scalable Infrastructure", Autonomous Agents conférence, 2001.
- Membre du comité de programme de la conférence internationale IADIS intitulée Applied Computing 2004, AC 2004, et de même pour l'édition de 2005
- Membre du comité de programme du workshop CoreGRID "Grid Systems, Tools and Environments", Octobre 2005, Grids@work conference.
- Membre du comité de programme du 3rd Workshop on Middleware for Grid Computing - MGC 2005 conjoint à ACM/IFIP/USENIX 6th International Middleware Conference, 2005. De même pour l'édition de 2006, conjoint à Middleware.
- Membre du comité de programme de la conférence internationale IASTED intitulée Parallel and Distributed Computing and Networks, PDCN 2004, et de même pour les éditions 2005, 2006 et 2007
- Membre du comité de programme du workshop *HPC Grid programming Environments and COmponents and Component and Framework Technology in High-Performance and Scientific Computing* (HPC-GECO+COMPFRAME), conjointement à HPDC 2006
- Membre du comité de programme de la 20ème édition de la conférence ECOOP (European Conference on Object-Oriented Programming), 2006

**Nationaux ou Francophones :**

- Membre du comité de programme de l'école d'hiver du groupe CAPA (Conception et Analyse d'Algorithmes Parallèles) du GDR PRS (Parallélisme - Réseaux - Systèmes), Janvier 1995
- Membre du comité de programme de l'école Parallélisme des groupes EXEC, CAPA et Rumeur du GDR PRS, pour décembre 1997
- Membre du comité de programme de la première édition des *Journées composants*, Grenoble, octobre 2002
- Membre du comité de programme de la conférence LMO (Langages et Modèles à Objets) 2003
- Membre du comité de programme de la troisième édition des *Journées composants*, Lille, mars 2004.
- Membre du comité de programme de la conférence LMO (Langages et Modèles à Objets) 2005.
- Membre du comité de programme des congrés francophones annuels RenPar7, RenPar8, RenPar9, RenPar10, RenPar11, RenPar12, RenPar13 (pour 2001), et RenPar16 (avril 2005) (Rencontres Francophones du Parallélisme, des Architectures et des Systèmes)

**Evaluations d'articles pour des comités de lecture ou de programme**

- Future Generation Computing Systems (FGCS), International European conference on Parallelism (EuroPar 2006) [2]
- ISPDC 2005, CompFrame 2005
- CMSGA at International Conference on Supercomputing (ICS) 2004, Book on Grid Computing : Software Environments and Tools
- International conference on Cluster Computing and the Grid (CCGrid), EuroPar'03, International conference on Compiler Construction (CC'03), Journal of Parallel and Distributed Computing (JPDC)
- International conference on High Performance Distributed Computing (HPDC'11), Joint ACM conference JavaGrande-ISCOPE, SIROCCO 2002, Cardis'02
- International Conference on SuperComputing (ICS'01), International Conference for High Performance Computing and Communications (SuperComputing SC'01), International conference on Autonomous Agents

[2]Les évaluations réalisées dans le cadre de la participation à des comités d'édition ou de programme ont été indirectement mentionnées précédemment, et ne sont donc pas repris ici. Chaque ligne de la liste correspond à une année

(AA'01), HPDC'10 , Journal on Mathematical Modelling and Numerical Analysis Modelisation (M2AN)
- Notere'2000, EuroPar'2000, I-SPAN'00, workshop JHPC at IPDPS 2000, Parallel Computing
- HPCN (High Performance Computing Network) Europe'99, ISCOPE'99
- ECOOP'97
- JPDC special issue on "Object-Oriented Real-Time Systems', OBPDC, Symposium on Theoretical Aspects of Computer Science (STACS'96), Euro-Par'96
- KBUP95

### 1.6.2 Collaborations scientifiques

**Nationales**

- INRIA[3] Sophia Antipolis,
  Equipe CAIMAN, coopération depuis 2002, sur la programmation parallèle haut niveau, par objets et composants, pour la résolution de problèmes de calcul scientifique. Coopération récemment soutenue par le projet de l'ANR, appel Calcul Intensif et Grilles de Calcul, *(Disco-Grid)*, notifié en décembre 2005.
  Equipe OMEGA, coopération depuis 2006, sur l'utilisation de technologies de programmation répartie sur grappes et grilles de calcul, pour résoudre des problèmes de mathématique financière. Coopération soutenue par le projet de l'ANR, appel Calcul Intensif et Grilles de Calcul, *Grilles de Calcul appliquées à des Problèmes de Mathématiques Financières (GCPMF)*, notifié en décembre 2005.
- Supelec, campus de Metz,
  Equipe SID (Stéphane Vialle), coopération depuis 2005, autour de la mise en œuvre parallèle et répartie de simulations dans le domaine de la finance, sur grilles de calcul. Soutenue par le projet GCPMF.
- LSR/IMAG,
  Equipe ADELE, coopération depuis 2003, autour du déploiement et de la supervision à grande échelle de passerelles OSGi. Coopération soutenue par le projet RNRT PISE, 2005-2006.

[3]Sont mentionnées ici les équipes avec lesquelles des travaux *communs* menant ou pouvant mener à des publications communes sont effectivement en cours, en général soutenus par des financements nationaux ou européens

**Internationales**

- Univ. of Pisa, Italie, Equipe de Marco Danelutto ; Univ. of Westminster, London, Equipe de Vladimir Getov ; Univ. of Münster, Equipe de Sergei Gorlatch ; CYFRONET, Krakow, Pologne, Equipe de Marian Bubak : coopérations bi-latérales sur la programmation par composants et par squelettes pour la grille, dans le cadre du réseau d'excellence CoreGRID, 2004-2008 ; elles ont vocation à être renforcées grâce au démarrage du projet européen GricComp (Grids Programming with Components : An Advanced Component Platform for an effective invisible grid) dont Denis Caromel est le leader scientifique (Juin 2006-2008).
- Universitad Federal Rio Grande Del Sul, Porto Allegre, Bresil, Equipe de Alexandre Navaud et Nicolas Maillard, coopération sur la programmation parallèle pour grappes et grilles. Coopération soutenue par la direction des relations internationales de l'INRIA, 2006.
- INRIA Grenoble, équipe SARDES (Sara Bouchenak), et UPM (Universidat Politécnica de Madrid), équipe LSD, (Marta Patiño-Martinez), coopération sur l'auto-administration de serveurs d'applications et leur déploiement en environnement de type grille. Coopération soutenue par l'ARC INRIA *Autonomic Management of Grid-Based Enterprise Services (AutoMan)*, 2006-2007
- ETSI (European Telecommunications Standardisation Institute), coopération autour des efforts de standardisation dans le cadre du grid computing, et participation aux réunions préparatoires de l'ETSI Technical Committee GRID (Mike Fisher (BT) Chairman), lancé officiellement en juin 2006.
- Fraunhofer Institute, Berlin ; Univ. Basel ; VTT Finland, coopération sur les architectures logicielles adaptatives et autonomes pour supporter les réseaux de communication de nouvelle génération, depuis Janvier 2006, dans le cadre du projet BioNets.

### 1.6.3 Contrats de Recherche

**Financements passés**

- Participation[4] au projet ESPRIT P440 MADS *Message-Passing Architectures and Description Systems*, 1988-1990
- Projet CNRS GDR PRS, Stratagème, de décembre 1993 à novembre 1995 (fin du projet). Laboratoires partenaires : PR*i*SM, LIFL, LIFO, LIP, LMC-IMAG, LAAS, ENSEEIHT, LaBRI.
- Projet ARCAD : Architecture Répartie extensible pour Composants ADaptables, projet exploratoire du RNTL, Novembre 2000 à Décembre 2002. Equipes partenaires : Rainbow (I3S CNRS), DTL/ASR (France Télécom R&D), SARDES (INRIA Rhône-Alpes), et OCM (Ecole des Mines de Nantes).
- Action Concertée Incitative (ACI) GRID 2001, "Globalisation des Ressources Informatiques et des Données", dans le cadre du projet "GRID RMI : Objets distribués haute performance pour la grille de calcul". Equipes partenaires : PARIS (IRISA), Runtime (LABRI), GOAL (LIFL), et CCR d'EADS (European Aeronautic Defence and Space Company). 2001-2002.
- Action Spécifique du CNRS, département STIC, du Réseau Thématique Pluriscipliniaire *Calcul à hautes performances et calcul réparti*, intitulée "Méthodologie de programmation des grilles : quelles directions de recherche dans les années à venir ?", 2004.
- Délégation française qui s'est rendue au Japon dans le cadre du workshop NII-Grid, 13-16 décembre 2004. Présentation des activités de l'équipe en terme de Grid computing, au Tokyo Institute of Technology.

**Financements en cours**

- Projet *PISE : Passerelle Internet Sécurisée et flExible* `www-adele.imag.fr/PISE`, projet pré-compétitif du RNRT, labellisé en mai 2003,

[4]Les contrats et financements listés ici, ne sont pas tous ceux dont mon équipe de recherche bénéficie ou a pu bénéficier, mais bien ceux autour desquels je me suis investie, notamment, en ce qui concerne le montage et/ou le suivi scientifique, administratif, financier, ou encore la production de rapports

démarré en janvier 2005 pour 2 ans. Laboratoires et entreprises partenaires : Schneider Electric SA, Université Joseph-Fourier - IMAG/LSR, France Télécom R&D, Trialog. Responsable scientifique de la participation de l'équipe INRIA OASIS à ce projet.

- Projet européen ITEA S4ALL *Services for All*, Juillet 2005- Juillet 2007. Le projet est construit autour de 3 ensembles de partenaires, soit 6 grandes companies (Alcatel CIT Research and Innovation, Bull, Nokia, Schneider Electric, Thales and Vodafone), 3 PMEs (Capricode, mCentric, Xquark), et 6 partenaires académiques (Fraunhofer Fokus, Helsinki Institute for Information Technology, INRIA, INT, Univ. Joseph Fourier IMAG, Univ. Politecnica de Madrid).
  Responsable scientifique de la participation de l'équipe INRIA OASIS à ce projet.
- Support technique à l'organisation du second Grid Contest and Plugtests, Grids@work conference, ETSI, financement e-Europe, sur la période avril-décembre 2005. Renouvellement d'un tel contrat pour l'organisation du troisième Grid Contest and Plugtests, Grids@work II, sur la période Juillet 2006-Janvier 2007.
- SSA (Support Specific Action) européene GridCoord *ERA Pilot on a co-ordinated Europe-wide initiative in Grid Research*, `www.gridcoord.org` Juillet 2004, pour 2 ans.
- Network of Excellence européen CoreGRID *Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies* `www.coregrid.net`, Sept. 2004 pour 4 ans. Implication dans les *Research Institutes* on *Programming Model*, and *Problem Solving Environment and Grid Systems*. Responsable administratif pour le partenaire INRIA Sophia-Antipolis/UNSA.
- Integrated Project BioNets : *Bio inspired Networking Technologies* `www.bionets.org` de l'unité FET (Future Emerging Technologies) de l'IST, Jan. 2006 pour 4 ans, conjointement avec l'équipe MAESTRO de l'INRIA Sophia-Antipolis. Participation à 6 tâches, dont 2 en tant que responsable.
- Projet de l'ANR, appel Calcul Intensif et Grilles de Calcul, *Grilles de Calcul appliquées à des Problèmes de Mathématiques Financières (GCPMF)*, Début 2006 pour 3 ans ; travaux conjoints en cours avec

l'équipe OMEGA de l'INRIA Sophia-Antipolis et Supelec.

- Projet de l'ANR, appel Calcul Intensif et Grilles de Calcul, *DIStributed objects and Components for high performance scientific computing on the GRID'5000 test-bed*, Début 2006, pour 3 ans, conjointement avec les équipes CAIMAN, SMASH de l'INRIA Sophia-Antipolis.
- Programme Relations Internationales de l'INRIA, International Students. Accueil de Elton Mathias, étudiant en master à l'UFRGS, Porto Allegre, Brésil, durant 6 mois de l'année 2006.
- ARC INRIA *Autonomic Management of Grid-Based Enterprise Services (AutoMan)* `http://sardes.inrialpes.fr/research/AutoMan/`, 2006-2007.
- Projet AGOS *Architecture Grid Orientée Services*, labellelisé dans le cadre du pôle de compétitivité de la région Provence Alpes Côte d'Azur, *Solutions Communicantes Sécurisées*. Partenariat avec Oracle, HP, Amadeus sur l'intégration des technologies de Grid aux services d'entreprise, mi juillet 2006 pour 3 ans.

### 1.6.4 Participations à des jurys de thèses de doctorat

- Houari Tine,
  "Notion d'état et Modèle objet : Réflexion et Contribution " , Institut National des Sciences Appliquees de Lyon, décembre 1999, en tant que co-rapporteur, avec D. Caromel.

- Eric Gascard,
  "Méthodes pour la vérification formelle de systèmes matériels et logiciels à architecture régulière", Université de Provence, juillet 2002, en tant qu'examinatrice.

- Pierre Vignéras,
  "Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones." Université de Bordeaux I, novembre 2004, en tant que co-rapporteur avec M. Riveill.

- Guillaume Mercier,

"Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques" Université de Bordeaux I, décembre 2004, en tant qu'examinatrice.

### 1.6.5 Encadrement de chercheurs

**Post-Doctorat**

- Nikolaos Parlavantzas, "Dynamic Software Components Composition in GRID Environments", mai 2005-février 2006. Co-encadrement avec Denis Caromel. Financement ERCIM dans le cadre du NOE CoreGRID.

**Doctorat**

- Nathalie Furmento, "SCHOONER : Une encapsulation orientée objet de support d'exécution pour applications réparties", soutenue en mai 1999. Co-encadrement à hauteur de 95%, avec Jean-Claude Bermond. Membres du jury : Jean-Paul Rigault (Président), Jean-François Méhaut, Bernard Vauquelin (Rapporteurs), Anne-Marie Pina-Dery.
  Nathalie Furmento a occupé ensuite un poste de "research associate" à Imperial College, Londres, puis a obtenu un poste d'ingénieur de recherche CNRS, au LaBri en janvier 2005.
- Fabrice Huet, "Objets mobiles : conception d'un middleware et évaluation de la communication". Co-encadrement à hauteur de 50%, avec Denis Caromel. Soutenue en décembre 2002. Membres du jury : Philippe Nain (Président), Brigitte Plateau, Jean-Bernard Stefani, Bernard Toursel (Rapporteurs).
  Fabrice Huet a été en 2003-2004 en post-doctorat INRIA à *vrije Universiteit d'Amsterdam*. Depuis septembre 2004, il occupe un poste de maitre de conférences à l'UFR Sciences de l'UNSA et est membre de l'équipe OASIS.
- Emmanuel Reuter, "Objets mobiles pour l'administration des systèmes et des réseaux". Encadrement à 100%. Membres du jury : Michel Riveill (Président), Olivier Festor, Serge Chaumette (Rapporteurs), J-L Ernandez.
  Début de thèse à temps partiel (emploi d'ingénieur système et réseau

à l'IUFM de Nice) en novembre 2000, soutenue en mai 2004. Depuis septembre 2005, mutation sur un poste d'ingénieur à l'Université de la Nouvelle Calédonie.
- Laurent Baduel, "Communications de groupe pour la programmation orientée objet parallèle". Co-encadrement à hauteur de 50%, avec Denis Caromel. Début de thèse en octobre 2001, soutenue en juillet 2005. Membres du jury : Johan Montagnat (Président), Henri Bal, El-Ghazali Talbi, André Schiper (Rapporteurs), Emmanuel Cecchet.
  Depuis novembre 2005, post-doctorant JSPS dans le groupe de Satoshi Matsuoka, Tokyo Institute of Technology.

**DEA/Master Recherche**

- Franck Delaplace, "Routage efficace de messages pour des processus migrants", DEA Informatique Paris 13, 1990-1991
- Nathalie Furmento, "Accès à PVM par l'intermédiaire du langage ADA dans un objectif de programmation conviviale d'applications parallèles", co-encadrement avec Daniel Lafaye de Micheaux, DEA Informatique UNSA, 1993-1994.
- Sylvain Gamel "Transmission efficace d'objets pour un langage orienté objet parallèle", DEA Informatique UNSA, 1995-1996.
- Fabrice Huet "Communication de groupe et migration", co-encadrement avec Denis Caromel, DEA Réseaux et Systèmes Distribués, UNSA, 1999.
- Emmanuel Reuter "Les agents mobiles et actifs pour l'administration de réseaux", DEA Réseaux et Systèmes Distribués, UNSA, 2000.
- Haris Saybasili, co-encadrement avec Denis Caromel, "Les Composants Hiérarchiques pour ProActive" DEA Réseaux et Systèmes Distribués, UNSA, 2002.
- Paul Naoumenko, co-encadrement avec Ludovic Henrio, "Component-oriented approach for adaptative and autonomic computing", Master STIC spécialité RSD et 3ème année ingénieur Polytech'Nice, UNSA, 2006. Poursuite en doctorat grâce à un financement avec la Région PACA.

**Stage Ingénieur**

- Călin Şandru, "Various Implementations of the MPI Standard : LAM, CHIMP, MPICH and UNIFY", Projet de fin d'étude de l'université de Timisoara, Roumanie, dans le cadre du projet TEMPUS, 1994-1995
- Philippe Barette et Cyrille Tonin, "Implémentation d'une librairie de communication pour un langage C++ parallèle au dessus de PVM et d'une bibliothèque de threads", Ecole Supérieure des Sciences Informatiques 3ème année, 1995-1996
- Ovidiu Codreanu, "Partitionnement de graphes pour de la simulation distribuée de réseaux de files d'attente" Projet de fin d'étude de l'université technique de Cluj-Napoca, Roumanie, dans le cadre du projet TEMPUS, 1995-1996
- Alexandre Clavaud, "Conception et développement d'un outil d'analyse de performances pour applications distribuées", Ecole Supérieure des Sciences Informatiques 3ème année, 1996-1997
- Damien Praca "Solution de transfert de données entre PC et téléphones mobiles pour les applications Java", Stage de fin détude, Ecole Supérieure d'Electronique et du Numérique de Lille. Co-encadrement avec V. Berge (Mobile-distillery), 2005.
- Frédéric Sassolas, "Princing of European Options with Distributed Monte Carlo simulations in ProActive", Stage Supelec 2ème année, 2005. Co-encadrement avec S. Vialle, Supelec.
- Doan Viet Dong et Samir Ouifi, "Grid computing applied to option pricing", Projet Master Professionel IMAFA, Polytech'Nice, UNSA, 2005-2006, co-encadrement avec Mireille Bossy, INRIA.
- Doan Viet Dong, "Parallélisation d'algorithmes en mathématique financière sur grille de calcul" , Stage Master Professionel IMAFA, Polytech'Nice, UNSA, 2005-2006, co-encadrement avec Mireille Bossy, INRIA.

# Chapitre 2

# Bibliographie personnelle

## Editions

[E1] R. Molva (organizer) and F. Baude (editor). Panel session : Mobile code, internet security and e-commerce. In J. Malenfant, S. Moisan, and A. Moreira, editors, *Workshop reader of ECOOP'2000*, number 1964 in LNCS, pages 270–281. Springer Verlag, 2000.

[E2] F. Baude, editor. *Calcul réparti à grande échelle - Meta-computing* (Ouvrage collectif). Hermès Science - Lavoisier, May 2002. 190 pages. ISBN 2-7462-0472-X.

[E3] M. Auguin, F. Baude, D. Lavenier, and M. Riveill, editors. *Actes de RenPar'15 - CFSE'3 - SympAAA'2003*. INRIA, oct 2003. 652 pages. ISBN 2-7261-1264-1.

[E4] F. Baude, editor. *Parallèlisme : Algorithmique, systèmes, applications*, volume 24(5) of *Techniques et Sciences Informatiques*. Hermès Lavoisier, 2005. Versions longues de communications sélectionnées lors de RenPar'15.

## Revues

[J1] F. Baude and D. Skillicorn. Vers de la programmation parallèle structurée fondée sur la théorie des catégories. *Technique et science informatiques*, 13 :494–525, 1994.

[J2] F. Baude, F. Belloncle, D. Caromel, N. Furmento, P. Mussi, Y. Roudier, and G. Siegel. Parallel object-oriented programming for parallel simulations. *Information Sciences, Elsevier Sc Pub.*, 93 :35–64, 1996.

[J3] F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Optimizing remote method invocation with communication-computation overlap. *Future Generation Computer Systems, Elsevier*, 18 :769–778, 2002. Selected article from PaCT 2001.

[J4] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Objets actifs mobiles et communicants. *Technique et science informatiques*, 21(6) :1–36, 2002.

[J5] F. Baude, D. Caromel, and D. Sagnol. Distributed objects for parallel numerical applications. *Mathematical Modelling and Numerical Analysis Modelisation, special issue on Programming tools for Numerical Analysis, EDP Sciences, SMAI*, 36(5) :837–861, 2002.

[J6] F. Baude, D. Caromel, C. Delbé, and L. Henrio. Un protocole de tolérance aux pannes pour objets actifs non préemptifs. *Technique et science informatiques*, 24(10), 823–847, 2005.

## Chapitres d'ouvrages :

[B1] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid (chapter 9). Springer, 2006. Pages 205–229. ISBN : 1-85233-998-5.

[B2] F. Baude, D. Caromel, F. Huet, T. Kielmann, A. Merzky, and H.Bal. *Future Generation Grids*, chapter Grid Application Programming Environments. CoreGRID series. Springer, jan 2006. ISBN : 0-387-27935-0. Also as the CoreGrid TR003 report, `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0003.pdf`.

[B3] J. Dünnweber, F. Baude, V. Legrand, N. Parlavantzas, S. Gorlatch *Towards Automatic Creation of Web Services for Grid Component Composition* Invited submission in the final proceedings of the 1st CoreGRID integration workshop, Volume 4 of the Springer CoreGRID proceedings series, 2006.

## Colloques internationaux avec sélection et actes

[C1] F. Baude, F. Carré, P. Cléré, and G. Vidal-Naquet. Topologies for large Transputer networks : Theoretical Aspects and Experimental Approach In *Proc. of the 10th Occam User Group Technical Meeting*, pages 178–197. Springfield, 1989.

[C2] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *8th Symposium of Theoretical Aspects of Computer Science, STACS*, pages 184–195. Springer-Verlag, 1991. LNCS 480.

[C3] N. Furmento and F. Baude. Schooner : An object-oriented run-time support for distributed applications. In *Proceedings of Parallel and Distributed Computing Systems (PDCS'96)*, volume 1, pages 31–36. International Society for Computers and their Applications (ISCA), Septembre 1996.

[C4] F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Overlapping Communication with Computation in Distributed Object Systems. In *Proc. of the 7th International Conference - High Performance Computing Networking'99, HPCN*, volume 1593 of *LNCS*, pages 744–753, April 1999.

[C5] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Communicating mobile active objects in java. In *Proc. of the 8th International Conference - High Performance Computing Networking, HPCN*, volume 1823 of *LNCS*, pages 633–643. Springer Verlag, May 2000.

[C6] F. Baude, A. Bergel, D. Caromel, F. Huet, O. Nano, and J. Vayssière. IC2D : Interactive Control & Debug for Distribution. In *3ème Int. Conference on "Large-Scale Scientific Computations"*, number 2179 of *LNCS*, 2001.

[C7] F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Optimizing Metacomputing with Communication-Computation Overlap. In *6th International Conference PaCT*, number 2127 in LNCS, pages 190–204. Springer Verlag, 2001.

[C8] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *11th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, pages 93–102. IEEE Computer Society, 2002.

[C9] E. Reuter and F. Baude. System and Network Management Itineraries for Mobile Agents. In *4th International Workshop on Mobile Agents for Telecommunications Applications, MATA*, number 2521 in LNCS, pages 227–238. Springer-Verlag, 2002.

[C10] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, 2002. ACM Press.

[C11] E. Reuter and F. Baude. A mobile-agent and SNMP based management platform built with the Java ProActive library. In *IEEE Workshop on*

*IP Operations and Management (IPOM 2002)*, pages 140–145, Dallas, 2002. ISBN 0-7803-7658-7.

[C12] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, LNCS, pages 1226–1242. Springer Verlag, 2003.

[C13] L. Baduel, F. Baude, D. Caromel, C. Delbe, N. Gama, S. El Kasmi, and S. Lanteri. A parallel object-oriented application for 3d electromagnetism. In *IEEE International Symposium on Parallel and Distributed Computing, IPDPS*, april 2004.

[C14] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Components for Numerical Grids. In *ECCOMAS 2004*, European Congress on Computational Methods in Applied Sciences and Engineering, Jyvaskyla, Finland, July 2004.

[C15] F. Baude, D. Caromel, and M. Morel. On Hierarchical, Parallel and Distributed Components for Grid Programming. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*, pages 97–108. Springer, June 2005. Proceedings of the Workshop on Component Models and Systems for Grid Applications held June 26, 2004 in conjunction with ACM International Conference on Supercomputing (ICS) 05.

[C16] L. Baduel, F. Baude, N. Ranaldo, and E. Zimeo. Effective and efficient communication in grid computing with an extension of proactive groups. International Workshop on Java for Parallel and Distributed Computing at IPDPS, 2005.

[C17] L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *CCGrid 2005 : IEEE/ACM International Symposium on Cluster Computing and the Grid*, Pages 824–831, Vol. 2. April 2005.

[C18] L. Baduel, F. Baude, D. Caromel, F. Huet, L. Henrio, S. Lanteri, and M. Morel. Grid Components Techniques : Composing, Gathering, and Scattering. In *Coupled Problems 2005, an ECCOMAS Thematic Conference*, European Congress on Computational Methods in Applied Sciences and Engineering, may 2005.

[C19] F. Baude, D. Caromel, C. Delbé, and L. Henrio. A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability. In *Proc. of the 11th International Euro-Par Conference*, volume 3648 of *LNCS*, pages 644–653. Springer-Verlag, 2005.

[C20] F. Baude and D. Caromel and M. Leyton and R. Quilici. Grid File Transfer during Deployment, Execution, and Retrieval. In *Proc. of the International Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06), part of OnTheMove Federated Conferences (OTM'06)*, to appear in the *LNCS*, Springer-Verlag, november 2006.

[C21] S. Bezzine, V. Galtier, S. Vialle, F. Baude, M. Bossy, D. Viet Dung, L. Henrio. *A Fault Tolerant Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing.* In *2nd IEEE International Conference on e-Science and Grid Computing.* To appear, December 2006.

## Colloques sans actes ou à diffusion restreinte :

[W1] F. Baude, S. Boucheron, and G. Vidal-Naquet. Equivalence entre les modèles de calcul parallèle pram et acteur. In *3èmes rencontres sur les algorithmes et architectures massivement parallèles*, oct 1990.

[W2] F. Baude. PRAM implementation on fine-grained mimd multicomputers. In *8th British Colloquim for theoretical computer science*, march 1992.

[W3] F. Baude, N. Furmento, and D. Lafaye de Micheaux. Managing true parallelism in ADA through PVM. In *1st European* PVM *Users' Group Meeting*, Rome, Italy, October 1994. `http://www.netlib.org/pvm3/epvmug94/`.

[W4] F. Baude, F. Belloncle, J.C. Bermond, D. Caromel, O. Dalle, E. Darrot, O. Delmas, N. Furmento, B. Gaujal, P. Mussi, S. Perennes, Y. Roudier, G. Siegel, and M. Syska. The SLOOP project : Simulations, parallel object-oriented languages, interconnection networks. In *2nd European School of Computer Science, Parallel Programming Environments for High Performance Computing ESPPE'96, pages 85-88, Alpe d'Huez, France*, april 1996.

[W5] F. Baude, N. Furmento, D. Caromel, R. Namyst, J.M. Geib, and J.F. Méhaut. C++// on top of $PM^2$ via SCHOONER. In *Proceedings of Stratagem'96*, pages 41–55, Sophia-Antipolis, France, July 1996. ISBN-2-7261-0982-9.

[W6] F. Baude, A. Bergel, D. Caromel, F. Huet, O. Nano, and J. Vayssière. IC2D : Interactive Control & Debug of Distribution. Grappes 2001, `http://www.univ-ubs.fr/valoria/grappes2001`, May 2001.

[W7] F. Baude, A. Bergel, D. Caromel, F. Huet, O. Nano, and J. Vayssière. Graphical Visualisation and Control of Distributed and Metacomputing Ap-

plications : IC2D. 1st EuroGlobus Workshop, http://www.euroglobus.unile.it/, June 2001. Invited talk.

[W8] F. Baude. Un exemple d'outil de programmation parallèle et distribuée orientée objets : ProActive. Réunion du club des utilisateurs du Calcul Parallèle sur le site de Sophia Antipolis et dans la région PACA, 24 avril, 2002. http://www.inria.fr/caiman/personnel/Stephane.Lanteri/clubPC-fr.html.

[W9] L. Baduel, F. Baude, and D. Caromel. Communications de Groupes Typés dans ProActive. In *École thématique sur la globalisation des ressources informatiques et des données (GRID)*, pages 74–84. CNRS, dec 2002.

[W10] F. Baude. ProActive : a Java library for parallel and distributed computing. Réunion du club des utilisateurs du Calcul Parallèle sur le site de Sophia Antipolis et dans la région PACA, 12 et 13 décembre, 2002. http://www.inria.fr/caiman/personnel/Stephane.Lanteri/clubPC-fr.html.

[W11] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. 3rd annual ObjectWeb conference, INRIA Rocquencourt, 20-21 novembre 2003. http://www.objectweb.org/conference2003/.

[W12] F. Baude, D. Caromel, and M. Morel. ProActive : An Open Source Middleware for Grid. 4th annual ObjectWeb conference, Lyon, January 2005. http://wiki.objectweb.org/ObjectWebCon05/.

[W13] F. Baude, D. Caromel, F. Huet, T. Kielmann, A. Merzky, and H.Bal. Grid Application Programming Environments : Comparing ProActive, Ibis, and GAT. In *Workshop on Grid Applications : from Early Adopters to Mainstream Users. In conjunction with GGF14, (June 27, Chicago, USA)*, June 2005. www.cs.vu.nl/ggf/apps-rg/meetings/ggf14.html.

[W14] F. Baude. Inria oasis's interests in grid standardization issues. 1st ETSI GRID standardisation meeting, Sept. 30th 2005.

[W15] S. Vialle, F. Sassolas, F. Baude, and V. Legrand. Distributed Financial Computations with ProActive. 2nd ProActive User Group, October 2005. www.inria.fr/oasis/plugtest2005/ProActiveUserGroup.html

[W16] F. Baude, L. Baduel, and D. Caromel. OO SPMD for the Grid : an alternative to MPI and the road to Component, October 2005. www.coregrid.net/mambo/content/view/188/30/

[W17] D. Puppin, M. Morel, D. Caromel, D. Laforenza, and F. Baude.

GRIDDLE Search for the Fractal Component Model. In *1st CoreGRID integration workshop*, Nov 2005.

[W18] F. Baude, V. Legrand, N. Parlavantzas, J. Dünnweber, and S. Gorlatch. Towards automatic creation of web services for grid component composition. In *1st CoreGRID integration workshop*, Nov 2005. Also selected and presented at the CoreGrid workshop : Grid Systems, Tools and Environments, Oct. 12th 2005, Grids@work conference.

[W19] F. Baude, D. Caromel, M. Leyton, and R. Quilici. Integrating deployment and file transfer tools for the grid. In *1st CoreGRID integration workshop*, Nov 2005. Also selected and presented at the CoreGrid workshop : Grid Systems, Tools and Environments, Oct. 12th 2005, Grids@work conference.

[W20] F. Baude. INRIA OASIS's interests in Grids : programming Grids 2nd ETSI GRID-WORKSHOP Standardisation meeting, Sophia-Antipolis, 24 May 2006.

[W21] F. Baude, *ProActive : Grid programming in the Services for All (S4ALL) perspective.* 2nd 2006 ObjectWeb architecture meeting, INRIA Lille, 12-13 June 2006.

[W22] F. Baude, D. Caromel, L. Henrio, M. Morel, P. Naoumenko. *Fractalising Fractal Controller for a Componentisation of the Non-Functional Aspects.* 5th Fractal Workshop in conjunction with ECOOP'20, Poster, July 2006.

[W23] F. Baude and A. Bottaro and J.M. Brun and A. Chazalet and A. Constancin and D. Donsez and L. Gurgen and P. Lalanda and V. Legrand and V. Lestideau and S. Marié and C. Marin and A. Moreau and V. Olive. *Extension de passerelles OSGi pour la grande échelle : Modèles et outils.* Atelier OSGi, 5 Septembre 2006, co-localisé avec UbiMob'06, 3ème Journées Francophones Mobilité et Ubiquité, Paris.

## Thèse, rapports techniques, divers

[R1] F. Baude. *Utilisation du paradigme acteur pour le calcul parallèle.* PhD thesis, Université Paris-Sud, Orsay, dec. 1991.

[R2] F. Baude. Pram implementation on fine-grained mimd multicomputers. Technical Report RR 220, Dept of computer science, Warwick University, UK, 1992.

[R3] F. Baude. Tree contraction on distributed memory parallel architectures. Technical report, Internal Research Report Queen's University, Kingston, Canada, 1992.

[R4] F. Baude, N. Furmento, and D. Lafaye de Micheaux. Managing true parallelism in ADA through PVM. Technical Report 94-62, Laboratoire CNRS I3S, 1994.

[R5] C. Sandru and F. Baude. Various implementations of the mpi standard : Lam, chimp, mpich and unify. Technical Report 95-22, Laboratoire CNRS I3S, 1995.

[R6] C. Roucairol, J. Roman, J.L. Roch, G. Villard, J.M. Geib, A. Ferreira, S. Miguet, F. Baude, B. Virot, G. Authié, J.M. Garcia, and P. Amestoy. Stratageme : une méthodologie de programmation parallèle pour les problèmes non structurés. rapport final. Technical report, PRiSM, Université de Versailles, 1995.

[R7] F. Baude and O. Dalle. Analyse des performances de communication du protocole pvm. Technical Report 96-08, Laboratoire CNRS I3S, 1996.

[R8] F. Baude. Coordination du deliverable d2.3 - "document de mise en œuvre". RNTL ARCAD, dec 2003.

[R9] F. Baude, D. Caromel, C. Delbé, and L. Henrio. A fault tolerance protocol for asp calculus : Design and proof. Technical Report RR-5246, INRIA, 2004.

[R10] the Oasis team. Report on ProActive User Group and Grid Plugtests. `www-sop.inria.fr/oasis/ProActive/plugtest_report.pdf`, 2004. GRID PLUGTEST : INTEROPERABILITY ON THE GRID at www.gridtoday.com, Vol 4, Issue 4, Jan. 2005.

[R11] the OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Roadmap version 1 on programming model. Technical Report D.PM.01, CoreGRID, Programming Model Virtual Institute, Feb 2005. 1st deliverable of the PM VI.

[R12] F. Baude and M. Morel. A study of the adequacy of the fractal model and tools for grid computing. Technical report, Internal technical report, CRE France Telecom R&D/INRIA, Apr 2005.

[R13] F. Baude, D. Caromel, L. Henrio, and M. Morel. Collective interfaces for a grid component model, proposed extensions to the fractal component model. Technical report, Internal technical report, CRE France Telecom R&D/INRIA, Nov 2005.

[R14] the OASIS team. Report from the 2nd grid plugtests and contest. www.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf, Dec 2005. Second Grid Plugtests Demo Interoperability at www.gridtoday.com, Vol 4, Issue 49, Dec. 2005.

[R15] F. Baude, D. Caromel, I. Rosenberg, and R. Quilici. The Use of Open Middleware for the Grids. Technical report, GridCoord SSA EU project, Dec 2005. Delivrable D.4.2.1, report from the GridCoord workshop organised at Grids@work.

[R16] the OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Proposals for a common grid component model for grid. Technical Report D.PM.02, CoreGRID, Programming Model Virtual Institute, Dec 2005. 2nd deliverable of the PM VI.

[R17] the OASIS team and other partners in the CoreGRID Programming Model Institute. Roadmap version 2 on programming model. Technical Report D.PM.03, CoreGRID, Programming Model Institute, Feb 2006. 3rd deliverable of the PM VI.

[R18] N. Parlavantzas, M. Morel, F. Baude, F. Huet, D. Caromel, and V. Getov. Componentising a scientific application for the grid. Technical Report TR-0031, CoreGRID, Institute on Grid Systems, Tools and Environments, Apr 2006. www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0031.pdf.

[R19] F. Baude, L. Henrio and partners VTT, TUB, UNIHH of the BIONETS consortium. Service Architecture Requirement Specification. BIONETS IP Project Delivrable from the Requirement and Analysis workpackage (3.1), July 2006. http://www.bionets.eu/BIONETS_D3_1_1.pdf

[R20] F. Baude, V. Legrand. PISE : Outil et API d'administration et déploiement à grande échelle. Livrable interne au projet RNRT PISE. https://gforge.inria.fr/docman/view.php/374/468/PISE%20doc(5).doc ou http://www.inria.fr/oasis/Francoise.Baude/PISEDeploiementAdminGrandeEchelle.pdf, July 2006.

[D1] F. Baude and N. Furmento. Schooner (version 1.0 de juin 1998). Dépot à l'agence de protection des programmes, 1998.
IDDN.FR.001.350017.00.R.P.1998.000.10600.

[D2] D. Caromel, F. Huet, J. Vayssière, F. Baude, A. Bergel, and O. Nano. ProActive (version 0.5.0 d'avril 2001). Dépot à l'agence de protection des programmes, 2001.

IDDN.FR.001.310003.00.R.P.2001.000.10600.

[D3] D. Caromel, F. Baude, L. Henrio, M. Malawski, M. Morel. GCM-CCA Interoperability. Working document. July 2006.

## Publications en cours de soumission

[S1] L. Baduel, F. Baude, D. Caromel. *Asynchronous Typed Object Groups for Grid Programming.* International Journal of Parallel Programming, May 2006.

[S2] N. Parlavantzas, M. Morel, F. Baude, F. Huet, D. Caromel, V. Getov. *Componentising a Scientific Application for the Grid* (voir [R18]). Submitted to the 2nd CoreGRID integration workshop. August 2006.

# Chapitre 3

# Bibliographie générale

# Bibliographie


[1] "cca specification". `http://cca-forum.org`.

[2] EGA Reference Model. Enterprise Grid Alliance, 2005.

[3] T. Abdellatif and M. Desertot. JOnAS 5, the ObjectWeb next generation Application Server. ObjectWeb Conference 2006.

[4] T. Abdellatif, J. Kornas, and J-B. Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In *Component Deployment*, volume 3798 of *LNCS*, pages 134–148, 2005.

[5] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9) :125–141, September 1990.

[6] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi, and C. Zoccolo. Autonomic Grid Components : the GCM Proposal and Self-optimising ASSIST Components. In *HPDC'15 workshop on HPC-GECO/Compframe*, 2006.

[7] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware assist applications via web services. In *Proc. of Intl. PARCO 2005 : Parallel Computing*.

[8] M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. In *Parallel Processing Letters*, 2002.

[9] M. Alt and S. Gorlatch. Using Skeletons in a Java-Based Grid System. In *Euro-Par*, number 2790 in LNCS, 2003.

[10] N. Andrade, L. Costa, G. Germoglio, and Walfredo Cirne. Peer-to-peer grid computing with the ourgrid community. In *23rd Brazilian Symposium on Computer Networks*, May 2005.

[11] BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase. Service Component Architecture - Building Systems

using a Service Oriented Architecture. Technical report, 2005. `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-sca/SCA_%White_Paper1_09.pdf`.

[12] Guy Bernard and Liela Ismail. Apport des agents mobiles à l'exécution répartie. *Technique et science informatiques*, 21(6) :771–796, 2002.

[13] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quéma, and J-B. Stéfani. "Architecture-Based Autonomous Repair Management : Application to J2EE Clusters". In *2nd IEEE Int. Conf. on Autonomic Computing (ICAC'05)*.

[14] H. Bouziane, C. Pérez, and T. Priol. Modeling and executing master-worker applications in component models. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, april 2006.

[15] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. An open component model and its support in java. In *International Symposium on Component-based Software Engineering*, 2004.

[16] J. Bustos-Jimenez, D. Caromel, and J. Piquer. Load Balancing : Towards the Infinite Network and Beyond. In *12th Workshop on Job Scheduling Strategies for Parallel Processing, in conjunction with SIGMETRICS 2006*.

[17] D. Caromel, C. Delbé, A. Di Costanzo, and M. Morel. Technical Services Components. In *HPDC'15 workshop on HPC-GECO/Compframe*, 2006.

[18] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive : an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology*, 12(1), 2006.

[19] D. Caromel and F. Huet. Un protocole adaptatif pour objets mobiles. In *Proc. of the 1st ACM French-speaking conference on Mobility and ubiquity computing*, 2004.

[20] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[21] H. Cervantes and R. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *26th International Conference on Software Engineering*. IEEE Computer Society, 2004.

[22] S-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu. Software architecture-based adaptation for grid computing. In *11th IEEE High Performance Distributed Computing*, July 2002.

[23] M. Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30, 2004.

[24] M. Danelutto. Second generation skeleton systems. Mynisymposium "Algoritmic Skeletons and High-level Concepts for Parallel Programming" associated with PARCO'05.

[25] M. Danelutto and P. Teti. Lithium : A Structured Parallel Programming Environment in Java. In *International Conference on Computational Science (ICCS)*, number 2330 in LNCS, 2002.

[26] P-C. David and T. Ledoux. Towards a Framework for Self-adaptive Component-based applications. In *DAIS*, number 2893 in LNCS. Springer Verlag, 2003.

[27] M. Desertot, C. Escoffier, and D. Donsez. Autonomic management of J2EE edge servers. In *MGC '05 : Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6. ACM Press, 2005.

[28] J. Dünnweber and S. Gorlatch. "HOC-SA : A grid Service Architecture for Higher-Order Components". In *Int. Conf. on Services Computing (SCC04)*.

[29] F. Exertier. Administration avancée de Jonas, la plate-forme J2EE ObjectWeb. Solutions Linux 2006. `www.solutionslinux.fr/document_conferencier/43f0d8ace9a64.pdf`.

[30] O. Festor and L. Andrey. *Standards pour la gestion des réseaux et des services*, chapter JMX : un standard pour la gestion Java. IC2 Réseaux et Télécoms. Hermès Science Publishing, 2004.

[31] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37 :70–82, 1996.

[32] G. Fox, D. Gannon, and M. Thomas, editors. *Special Issue : Grid Computing Environments*, volume 14, 2002.

[33] S. Frenot and D. Stefan. Instrumentation de plates-formes de services ouvertes - Gestion JMX sur OSGi. In *Mobile and Ubiquitous Computing 2004, June 1-3, 2004*. ACM Press.

[34] M. Fukuda, Y. Tanaka, N. Suzuki, L. Bic, and S. Kobayashi. A Mobile-Agent-Based PC Grid. In *IEEE Autonomic Computing Workshop*, 2003.

[35] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. "On Building Parallel & Grid Applications : Component Technology and Distributed Services". In *Second International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 04)*.

[36] Li Gong. A software architecture for open service gateways. *IEEE Internet Computing*, 5(1) :64–70, 2001.

[37] S. Gorlatch. Send-receive considered harmful : Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1), 2004.

[38] T. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2), 1994.

[39] F. Huet, D. Caromel, and H. Bal. A High Performance Java Middleware with a Real Application. In *Proc. of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing*.

[40] ObjectSpace Inc. "ObjectSpace Voyager Technical Overview". `http://www.recursionsw.com/whitepapers.htm`, 1997.

[41] INMOS. Transputer. `http://en.wikipedia.org/wiki/Transputer`.

[42] JCP. Java Business Integration (JBI). `http://www.jcp.org/en/jsr/detail?id=208`, 2005.

[43] R. Jimenez, M. Patiño, and B. Kemme. Enterprise Grids : Challenges Ahead. In *Int. Workshop on High-Performance Data Management in Grid Environments co-located with VECPAR 7th Int. Conf. on High Performance Computing for Computational Science*, 2006.

[44] N. Karonis, B. Toonen, and I. Foster. MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5), 2003.

[45] P. Kuonen, G. Babin, N. Abdennadher, and P-J. Cagnard. Intensional High Performance Computing. In *Workshop on Distributed Communities on the Web (DCW 2000)*, number 1830 in LNCS. Springer-Verlag, 2000.

[46] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam. Towards Self-Organizing Distributed Computing Frameworks : The H2O Approach. *Parallel Processing Letters*, 13(2), 2003.

[47] P. Lalanda, A. Chazalet, and V. Lestideau. Deployment of software services in the power distribution context. In *4th IEEE Int. Conf. on Industrial Informatics (INDIN'06)*.

[48] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 24(3), 1999.

[49] M. Lewis, A. Ferrari, M. Humphrey, J. Karpovich, M. Morgan, A. Natrajan, A. Nguyen-Tuong, G. Wasson, and A. Grimshaw. Support for extensibility and site autonomy in the Legion grid system object model. *Journal of Parallel Distributed Computing*, 63, 2003.

[50] H. Liu, M. Parashar, and S. Hairiri. A component-based programming framework for Autonomic Applications. In *1st IEEE Int. Conf. on Autonomic Computing (ICAC-04)*.

[51] S. Loke. Towards Data-Parallel Skeletons for Grid Computing : An Itinerant Mobile Agent Approach. In *Int. IEEE/ACM Symposium on Cluster Computing and the Grid (CCGRID'03)*, 2003. Poster.

[52] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. "Alchemi : A .NET-Based Enterprise Grid Computing System". In *Proc. of the 6th Int. Conference on Internet Computing (ICOMP'05)*.

[53] M. Malawski, D. Kurzyniec, and V. Sunderam. MOCCA - Towards a Distributed CCA Framework for Metacomputing. In *HIPS*, 2005.

[54] C. Marin, N. Belkhatir, and D. Donsez. Gestion transactionnelle de la reprise sur erreurs dans le déploiement. In *Proceedings of the 1ère Conférence Francophone sur le Déploiement et la (Re)Configuration de Logiciels (DECOR'2004)*. `http://hal.ccsd.cnrs.fr/DECOR04`.

[55] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 2004.

[56] K. Matsuzaki, Z. Hu, and M. Takeichi. Implementation of Parallel Tree Skeletons on Distributed Systems. In *Proceedings of the Third Asian Workshop on Programming Languages and Systems (APLAS '02)*.

[57] Ernst W. Mayr and Ralph Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distributed Computing*, 26(2) :181–192, 1995.

[58] V. Mencl and T. Bures. "Microcomponent-Based Component Controllers : A Foundation for Component Aspects". In *12th Asia-Pacific Software Engineering Conference (APSEC 2005)*.

[59] N. Merle. *Architecture pour les systèmes de déploiement logiciel à grande échelle : prise en compte des concepts d'entreprise et de stratégies*. PhD thesis, Univ. Joseph Fourier de Grenoble, 2005.

[60] M. Migliardi and V. Sunderam. The HARNESS PVM-Proxy : Gluing PVM Applications to Distributed Object Environments and Applications. In *Proceedings of the 9th Heterogeneous Computing Workshop*, 2000.

[61] "MIT Artificial Intelligence Laboratory". The Jellybean Machine. `http://cva.stanford.edu/projects/j-machine/`.

[62] R. Namyst and J.F. Méhaut. "PM$^2$ : Parallel Multithreaded Machine. A Computing environment for distributed architectures". In *Int. Conference on Parallel Computing (ParCo'95)*, 1995.

[63] T. Nguyen and P. Kuonen. ParoC++ : A Requirement-driven Parallel Object-oriented Programming Language. In *International Conference on Computational Science (ICCS)*, LNCS, 2003.

[64] R. Nieuwpoort, J. Maassen, G. Wrzesinacuteska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis : a flexible and efficient java-based grid programming environment. *Concurrency and Computation : Practice and Experience*, 17(7–8), 2005.

[65] L. Pan, L. Bic, M. Dillencourt, and M. Lai. From distributed sequential computing to distributed parallel computing. In *5th Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-03)*. `www.ics.uci.edu/~bic/messengers/papers/HPSECA03.pdf`.

[66] M. Pasin, P. Kuonen, M. Danelutto, and M. Aldinucci. Skeleton Parallel Programming and Parallel Objects. In *Proceedings of the 1st CoreGRID Integration Workshop*, 2005.

[67] A. Puliafito and O. Tomarchio. Using mobile agents to implement flexible network management strategies. *Computer Communication Journal, 23(8) :708–719*, 2000.

[68] A. Ranade. How to emulate shared memory. *J. of Computer and System Sciences*, 1991.

[69] Ichiro Satoh. Building reusable mobile agents for network management. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 33(3) :350–357, 2003.

[70] S. Schaefer. Configuration, description, deployment, and lifecycle management (ccdml) - component model version 1.0. Technical report, 2006. `http://www.ggf.org/documents/GFD.65.pdf`.

[71] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with Components and Aspects. In *9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, number 4063 in LNCS.

[72] D. Skillicorn. Architecture-independant parallel computation. *IEEE Computer*, 23(12) :38–51, Dec 1990.

[73] D. Skillicorn. *Foundations of Parallel Computing*. Cambridge University Press, 1994.

[74] D. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39 :115–125, Jul 1996.

[75] D. Skillicorn. Structured Parallel Computation in Structured Documents. *J. Universal Computer Science*, 3(1), 1997.

[76] D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1) :65–83, Jul 1995.

[77] D. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising Data-Parallel Programs Using the BSP Cost Model. In *EuroPar Conference*, 1998.

[78] A. Streit, D. Erwin, T. Lippert, D. Mallmann, R. Mendey, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and P. Wieder. UNICORE - From Project Results to Production Grids. *Grid Computing : New Frontiers of High Performance Computing*, 2005.

[79] H. Hoang To, S. Krishnaswamy, and B. Srinivasan. Mobile agents for network management : when and when not ! In *SAC '05 : Proceedings of the 2005 ACM symposium on Applied computing*, pages 47–53. ACM Press, 2005.

[80] O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *2nd Workshop on Active Middleware Services (AMS'00) in HPDC-9, August 2000.* `citeseer.ist.psu.edu/tomarchio00active.html`.

[81] S. Vadhiyar and J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation : Practive and Experience*, 17, 2004.

[82] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 1990.

[83] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. "Active Messages : a Mechanism for Integrated Communication and Computation". In *19th ACM Int. Symposium on Computer Architectures*, 1992.

[84] A. Zain, P. Trinder, H-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *Journal of Scalable Computing : Practice and Experience*, 7(3), 2006.

[85] L. Zhang and M. Parashar. Enabling efficient and flexible coupling of parallel scientific applications. In *IEEE International Symposium on Parallel and Distributed Computing, IPDPS*, april 2006.